Titre: Title:	Évaluation de l'impact des erreurs numériques dans un logiciel de calcul de dose en radiothérapie par la méthode de Monte-Carlo sur GPU
Auteur: Author:	Vincent François Magnoux
Date:	2014
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Magnoux, V. F. (2014). Évaluation de l'impact des erreurs numériques dans un logiciel de calcul de dose en radiothérapie par la méthode de Monte-Carlo sur GPU [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1520/

UNIVERSITÉ DE MONTRÉAL

ÉVALUATION DE L'IMPACT DES ERREURS NUMÉRIQUES DANS UN LOGICIEL DE CALCUL DE DOSE EN RADIOTHÉRAPIE PAR LA MÉTHODE DE MONTE-CARLO SUR GPU

VINCENT FRANÇOIS MAGNOUX
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2014

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ÉVALUATION DE L'IMPACT DES ERREURS NUMÉRIQUES DANS UN LOGICIEL DE CALCUL DE DOSE EN RADIOTHÉRAPIE PAR LA MÉTHODE DE MONTE-CARLO SUR GPU

présenté par : MAGNOUX Vincent François
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. DAGENAIS Michel, Ph.D., président
M. OZELL Benoît, Ph.D., membre et directeur de recherche
M. DESPRÉS Philippe, Ph.D., membre et codirecteur de recherche
M. PAL Christopher J., Ph.D., membre

REMERCIEMENTS

J'aimerais offrir des remerciements particuliers à Benoît Ozell et à Philippe Després pour m'avoir donné l'opportunité de travailler sur ce projet, pour avoir passé des heures à en discuter, me l'avoir fait retravailler et ainsi m'avoir permis de le mener à terme. Je voudrais également les remercier pour leur aide à la rédaction de mon premier article et à la préparation de ma première présentation à une conférence.

RÉSUMÉ

Cette étude se penche sur les erreurs issues de la représentation imparfaite des nombres réels par les ordinateurs dans le cadre de programmes de calcul scientifique, plus particulièrement ceux exécutés sur carte graphique (GPU). Ces erreurs se produisent parce que les nombres manipulés constituent seulement une approximation des nombres réels, qui doivent donc être arrondis. En général, les calculs sont faits avec une précision suffisante pour donner les résultats attendus, mais il arrive que les erreurs d'arrondi s'accumulent ou se combinent pour donner des résultats inexacts. L'objectif de ce travail est de déterminer, à l'aide d'une série de tests, si de telles erreurs sont présentes dans un programme spécifique sur GPU, `bGPUMCD`.

`bGPUMCD` est un programme utilisé en radiothérapie pour simuler la quantité de radiation distribuée à un patient. La simulation est basée sur une méthode de Monte-Carlo, c'est-à-dire qu'un grand nombre de particules sont simulées individuellement, avec leur trajectoire et leurs interactions, et l'énergie qu'elles déposent en chaque point du volume simulé est enregistrée. Pour calculer le dépôt d'énergie, le volume est divisé en une grille régulière de voxels, qui contiennent chacun l'énergie donnée par toutes les particules qui les ont traversés.

Pour évaluer la précision des résultats de `bGPUMCD`, trois aspects des calculs sont testés : la précision utilisée (simple ou double), l'implantation des fonctions arithmétiques (logicielle ou matérielle) et la dimension des voxels. Deux composantes de `bGPUMCD` sont également analysées plus en profondeur : l'accumulation de l'énergie dans chaque voxel et le traçage de particule. Pour faciliter, de façon générale, l'analyse de programmes pour des erreurs numériques, une méthode automatisée de détection d'erreurs est développée.

Les résultats révèlent la présence d'erreurs importantes dans le processus d'accumulation d'énergie dans les voxels entourant les sources de radiation. Lorsque l'énergie atteint une certaine quantité, les contributions individuelles des particules sont trop petites pour avoir un effet lors d'une addition en précision simple. Par ailleurs, une fonction importante pour le calcul de la trajectoire des particules retourne dans quelques cas rares un résultat erroné. Les autres aspects étudiés, soient l'implantation particulière des fonctions arithmétiques et la discrétisation en voxels, ne semblent pas causer d'erreurs majeures. Finalement, la méthode automatique d'analyse mise au point permet de détecter les erreurs trouvées par d'autres moyens au cours de l'étude.

Des erreurs d'origine numérique se produisent en effet dans `bGPUMCD`, sous des conditions très spécifiques. Certaines de ces erreurs sont facilement réparables, par exemple en changeant l'ordre de certaines opérations, tandis que d'autres nécessitent des changements majeurs

dans les algorithmes utilisés. Les types d'erreurs détectées sont très génériques et peuvent se retrouver dans de nombreux programmes de calcul scientifique, qu'ils soient basés sur des méthodes de Monte-Carlo ou non ; nos résultats peuvent donc servir d'exemples pour certaines situations auxquelles il est nécessaire de porter une attention particulière lors du développement d'une application sur GPU.

ABSTRACT

This study examines errors produced by the representation of real numbers by processors in the context of a scientific program executed on a graphics processor (GPU). This representation only approximates real numbers and thus contains a rounding error. In most cases, this error is too small to significantly alter the computed values, but in some situations it can accumulate in such a way that the results become inaccurate or meaningless. The purpose of this work is to design a series of tests that will help determine whether such errors occur in a specific GPU-based program, `bGPUMCD`.

`bGPUMCD` is used in radiation therapy to simulate the radiation distribution received by a patient. This simulation is based on Monte Carlo methods, by which a large number of particles are simulated individually to determine the amount of energy they deposit in every part of the simulated volume. The volume is divided into voxels for the purpose of energy scoring.

To provide a general idea of the precision of `bGPUMCD`'s results, three aspects of the computations were tested: their precision (single or double), the implementation used for arithmetic functions (software of hardware) and voxel size. Additionally, two components of `bGPUMCD`, energy accumulation per voxel and particle tracking, were further tested. Finally, an automatic error detection method was implemented in order to provide an easier way to analyse programs for numerical errors.

Major errors were detected in the energy accumulation process, in the voxels surrounding radiation sources. When the energy in a voxel reaches a certain threshold, contributions from individual particles are too small to be taken into account during a single-precision addition. In the particle tracking component, a function was found to return a meaningless value in some rare cases. The other tests indicate no major errors in other aspects of the computations (arithmetic functions and discretization). The automatic analysis method was able to detect the errors found by other means during the study.

Numerical errors do occur in `bGPUMCD`, under very specific conditions. Some can be avoided by simply reordering some operations, while others require major changes in the algorithm. The detected errors arise from very simple computations which can be found in many program other than `bGPUMCD`, which may or may not be Monte Carlo simulations. When precision is important, careful consideration must be given to such possibilities.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

CPU	*Central processing unit*, processeur
DSA	*Discrete stochastic arithmetic*, arithmétique stochastique discrète
FMA	*Fused multiply-add*, opération multiplication-addition
GPU	*Graphics processing unit*, processeur graphique
NaN	*Not a Number*, résultat d'une opération arithmétique invalide en virgule flottante
ulp	*Unit in the last place*, unité correspondant à la valeur du dernier bit d'un nombre à virgule flottante

CHAPITRE 1

INTRODUCTION

Le travail présent vise à évaluer la précision numérique des calculs effectués sur carte graphique (ou GPU) à travers l'étude d'un programme spécifique, `bGPUMCD` (Hissoiny *et al.*, 2011a). L'utilisation croissante des cartes graphiques dans des applications de calcul scientifique rend importante la question de leur précision, par rapport à celle des processeurs (CPU) généralement utilisés. Les GPU permettent d'effectuer une grande quantité de calculs simultanément, mais peuvent parfois sacrifier une partie de la précision de ces calculs pour en augmenter la vitesse.

Le prochain chapitre présente les types d'erreurs pouvant survenir lors des calculs faits par un processeur, graphique ou non, ainsi que quelques particularités des GPU. Il présente également une revue des moyens utilisés pour détecter et corriger ces erreurs. Les objectifs et hypothèses de ce travail sont décrits à la fin de ce chapitre. Le chapitre 4 (introduit par le chapitre 3) présente une partie de la démarche et des résultats permettant d'atteindre ces objectifs et a été soumis pour publication comme article scientifique. Le reste de la démarche est détaillé au chapitre 5. Les résultats sont présentés au chapitre 6 et une dicussion en est présentée au chapitre 7.

CHAPITRE 2

REVUE DE LITTÉRATURE

2.1 Introduction

Pour manipuler les nombres réels à l'aide d'un ordinateur, il est nécessaire de les représenter avec une quantité finie d'information, sous forme de bits. Puisqu'il existe une quantité infinie de ces nombres – même lorsque des bornes sont imposées – une discrétisation est nécessaire, qui entraîne une approximation des nombres représentés. Cette approximation donne lieu à une petite erreur qui, dans la majorité des cas, n'influence pas de façon significative les résultats d'un calcul. Cependant, pour des applications comportant une grande quantité de calculs ou dans certains cas particuliers, cette erreur peut s'accumuler ou être accentuée de façon extrême et ainsi entraîner des résultats entièrement différents de la quantité calculée. Cette erreur inhérente à la représentation approximative des nombres réels sera décrite plus loin, ainsi que différentes manières dont elle peut influencer le résultat d'un programme.

Parmi les représentations possibles, la virgule flottante, selon la norme IEEE-754, est la plus utilisée par les processeurs généraux et par les cartes graphiques NVIDIA (Whitehead et Fit-Florea, 2011). De manière semblable à la notation scientifique, les nombres sont caractérisés par un coefficient normalisé – qui comporte exactement un chiffre avant la virgule – ainsi qu'un exposant appliqué à la base utilisée. Les aspects des nombres à virgule flottante et de la norme IEEE-754 pertinents à cette étude seront tout d'abord décrits, puis certaines difficultés reliées à cette représentation seront illustrées à l'aide d'exemples simples.

La source première d'information sur les nombres à virgule flottante est la norme IEEE-745 elle-même (IEEE, 2008), puisque c'est celle qui est utilisée par les processeurs et les cartes graphiques. Elle décrit notamment le format sous lequel sont représentés les nombres réels, les modèles possibles pour arrondir le résultat d'un calcul et la manière dont certaines opérations mathématiques doivent être effectuées.

Plusieurs formats sont définis par la norme, selon la quantité de bits et la base utilisés. Les formats binaires les plus communs – les seuls utilisés par les GPU NVIDIA – sont la précision simple (32 bits), contenant 8 bits pour encoder l'exposant et 23 pour le coefficient, et la précision double (64 bits), avec 11 bits pour l'exposant et 52 pour le coefficient. Lorsqu'un nombre à virgule flottante en base binaire est normalisé, le chiffre avant la virgule est nécessairement 1 ; il n'est donc pas nécessaire de l'encoder avec le reste du coefficient, ce qui permet d'économiser un bit. La véritable précision pour les virgules flottantes simple et double est

donc respectivement de 24 et 53 bits. L'inconvénient de cette notation est qu'elle laisse un espace relativement grand entre zéro et le plus petit nombre supérieur à zéro représentable. La solution adoptée par la norme IEEE est de considérer le chiffre avant la virgule comme 0 lorsque l'exposant est minimal ; la représentation est alors considérée comme "dénormalisée".

La norme définit quatre manières d'arrondir le résultat d'un calcul impliquant des virgules flottantes : vers le haut (le plus petit nombre supérieur), vers le bas (le nombre inférieur le plus élevé), vers zéro (le nombre dont la valeur absolue est la plus grande, tout en étant inférieure à celle du résultat) ou vers le nombre le plus près. Dans le dernier cas, lorsqu'un résultat tombe exactement entre deux nombres représentables, il doit être arrondi vers le nombre pair. La plate-forme CUDA permet de choisir parmi ces modes dans le code source en ajoutant *_mode* à la fin des opérations, où *mode* peut être *ru*, *rd*, *rz* ou *rn*. Par défaut, les opérations sont arrondies au plus près. La norme stipule par ailleurs que les opérations mathématiques de base – addition, soustraction, multiplication, division – ainsi que la racine carrée doivent donner comme résultat la quantité comme si elle avait été calculée avec une précision infinie, puis arrondie selon une des règles possibles.

Goldberg (1991) présente un survol plutôt détaillé des problèmes potentiels reliés à l'utilisation de virgule flottante. La source principale de ces problèmes est l'erreur contenue dans un nombre à virgule flottante lorsqu'il décrit un nombre réel qui n'est pas représentable avec la notation utilisée. Cette situation se produit notamment lorsque le nombre réel se situe entre deux nombres à virgule flottante ; par exemple, le nombre décimal 0.1_{10} devient un nombre périodique (0.000110011_2...) en représentation binaire. L'autre situation dans laquelle un nombre ne serait pas représentable est s'il se situe au-delà des valeurs possibles pour un certain format ; par exemple, un nombre pourrait être plus grand que le plus grand nombre à virgule flottante représentable.

Les mesures les plus communes de l'erreur contenue dans un nombre à virgule flottante sont la quantité d'unités de plus petit poids (*units in the last place*, ou *ulp*) et l'erreur relative. Les unités de plus petite précision mesurent l'erreur sur la dernière unité du nombre représenté. Lorsque ce nombre est arrondi au plus près, l'erreur maximale est de 0.5 ulp ; dans les autres modes, elle est de 1 ulp. L'erreur relative mesure quant à elle l'erreur par rapport à la taille du nombre. En précision simple, l'erreur relative correspondant à 0.5 ulp se situe entre 2^{-24} et 2^{-23}. La plus grande de ces valeurs correspond à l'epsilon machine ε, soit la borne supérieure sur l'erreur de représentation en virgule flottante.

Comme le décrit Higham (2002), les sources possibles d'erreurs dans un programme de calcul sont l'arrondi, la troncature et les erreurs contenues dans les données. Les erreurs d'arrondi proviennent de la représentation discrète des nombres réels et de l'arrondi des résultats des divers calculs présents dans une application. L'erreur de troncature provient de l'utilisa-

tion de sommes finies pour calculer de façon approximative une fonction – par exemple, les fonctions trigonométriques. Finalement, les erreurs dans les données proviennent principalement de l'incertitude des instruments utilisés pour les collecter ou de leur numérisation. Seule l'erreur d'arrondi sera discutée dans cette étude ; l'erreur de troncature ne sera pas traitée directement, mais sera plutôt considérée comme une variante de l'erreur d'arrondi.

La manipulation de nombres arrondis plutôt qu'exacts signifie que plusieurs propriétés arithmétiques d'une expression, comme l'associativité, ne sont plus valides. L'expression $(a + b) + c$ n'est donc pas équivalente à $a + (b + c)$. En choisissant par exemple $a = 1 \times 10^{30}$, $b = -1 \times 10^{30}$ et $c = 1$, le résultat de la première expression est 1, tandis que la seconde donne 0, pour des calculs faits en précision simple ou double. Il faut donc porter une attention particulière à l'ordre dans lequel s'effectuent les calculs, ce qui peut être problématique sur une plate-forme parallèle comme une carte graphique, où il n'y a pas de moyen efficace de contrôler directement l'ordre d'exécution des différents fils.

L'erreur contenue dans un nombre à virgule flottante ne cause pas nécessairement de résultat aberrant. Cependant, cette erreur peut être amplifiée par annulation (*cancellation*). Lorsque deux quantités proches sont soustraites, leurs chiffres les plus significatifs s'annulent. Si ces deux quantités sont représentées exactement, le résultat sera également exact. Par contre, si elles contiennent une certaine erreur, cette erreur est augmentée de façon exponentielle, selon la proximité des deux nombres. Parfois, une telle annulation est évitable en réarrangeant les termes d'une équation. L'utilisation d'un nombre contenant une erreur amplifiée peut affecter grandement le résultat d'un programme, en particulier si le flot d'exécution est modifié lors d'une comparaison avec un tel nombre.

2.1.1 Particularités des cartes graphiques NVIDIA

En plus des opérations mathématiques de base et de la racine carrée, les cartes graphiques NVIDIA fournissent aussi une opération multiplication-addition (*fused multiply-add*), de forme $a + b * c$, qui comporte un seul arrondi pour les deux opérations effectuées, plutôt qu'un pour la multiplication et un pour l'addition (NVIDIA, 2012).

Un autre aspect qui mérite mention est l'option *flush-to-zero* fournie par `nvcc`, le compilateur de CUDA. Lorsque cette option est activée, un calcul qui résulte en un nombre dénormalisé est simplement arrondi à zéro, ce qui accélère le traitement au prix d'une perte de précision. Une conséquence de cette élimination des nombres dénormalisés est que lorsque deux nombres différents sont très rapprochés, leur différence peut être égale à zéro.

Enfin, les GPU NVIDIA fournissent l'option d'effectuer certains calculs à l'aide de fonctions intrinsèques. Ces fonctions sont une implantation matérielle des fonctions disponibles à travers la librairie mathématique de CUDA auxquelles elles correspondent. Elles s'exécutent

beaucoup plus rapidement qu'avec l'implantation logicielle, mais utilisent des algorithmes différents, légèrement moins précis, qui ne se conforment pas à la norme IEEE. Le guide de programmation CUDA (NVIDIA, 2012) décrit exactement en quoi les résultats des fonctions intrinsèques divergent de la norme IEEE. Ces différences sont généralement limitées au dernier bit du nombre calculé, pour certaines valeurs spécifiques en entrée à la fonction.

2.2 Le logiciel `bGPUMCD`

Le programme qui sera examiné dans cette étude, `bGPUMCD`, est une variante de `GPUMCD`, un programme présenté initialement par Hissoiny *et al.* (2011a). Il sert à calculer la dose de radiation reçue par un patient dans le cadre d'une radiothérapie, à l'aide de la méthode de Monte-Carlo. Pour ce faire, les particules qui déposent la dose sont simulées individuellement. Leur trajectoire et leurs interactions sont déterminées de façon aléatoire, selon des probabilités connues. Plus le nombre de particules simulées est grand, plus la dose totale calculée s'approche de la réalité – l'incertitude sur le résultat est inversement proportionnelle à la racine carrée du nombre de particules simulées. `bGPUMCD` a été développé spécifiquement pour des applications en curiethérapie (Hissoiny *et al.*, 2012) et ne simule que des photons, contrairement au programme original `GPUMCD`, qui simule également les électrons générés par les photons.

Pour déterminer la distribution de la dose, le volume simulé est divisé en voxels – un équivalent en trois dimensions de pixels. En tout temps, les particules se trouvent dans un certain voxel et y déposent une énergie proportionnelle à la distance parcourue dans celui-ci. Cette technique de simulation de radiation est connue sous le nom d'estimateur de parcours linéaire (Williamson, 1987).

La modélisation des sources de particules constitue une exception à la discrétisation en voxels. Les sources sont de petits bâtonnets radioactifs qui sont directement implantés dans un patient pour le traitement. La *selectSeed* ^{125}I par exemple, une des sources dont la description est implantée dans `bGPUMCD`, est un cylindre d'argent recouvert d'une couche d'iodure d'argent, imbriqué dans une capsule de titane (Karaiskos *et al.*, 2001). Les différentes couches de matériaux qui composent les sources sont décrites par des équations de surfaces quadriques (une série de cylindres imbriqués les uns dans les autres). Cette description permet une précision beaucoup plus grande que les voxels, limitée seulement par la précision de la représentation des nombres réels utilisés.

Les sources potentielles d'erreur les plus évidentes dans un tel programme sont l'accumulation de la dose dans chaque voxel et le traçage de rayon. Dans le premier cas, l'erreur peut tout simplement s'accumuler avec le temps, puisque le nombre de particules simulées peut

atteindre plusieurs milliards ; par ailleurs, puisque l'ordre dans lequel les doses sont ajoutées à un voxel peut varier lorsque plusieurs fils tentent d'effectuer l'addition simultanément, les résultats ne sont pas les mêmes d'une exécution à l'autre – même sans l'aspect aléatoire de la simulation. Dans le second cas, une petite erreur d'arrondi peut avoir des effets relativement grands lorsqu'elle place une particule dans le mauvais voxel ou qu'elle la fait dévier tranquillement de sa trajectoire. Ces effets peuvent être modérés par la taille des voxels ou l'implémentation d'une géométrie non voxélisée.

2.3 Méthodes d'analyse

L'approche la plus simple pour résoudre les problèmes liés aux nombres à virgule flottante consiste à faire les calculs avec une précision supérieure. Cependant, pour détecter et corriger des erreurs numériques dans un programme, il est nécessaire de procéder à son analyse, soit mathématiquement, à la main, soit de façon automatique. L'analyse automatique peut être effectuée statiquement, sur le code source, ou dynamiquement, lors de l'exécution du programme.

2.3.1 Augmentation de la précision

En augmentant la précision des nombres utilisés – de simple à double, ou à quadruple, ou même à une précision arbitraire – les erreurs d'arrondi sont réduites de façon importante. Ainsi, un programme dont la seule source d'erreur dans les résultats est une accumulation des erreurs d'arrondi peut augmenter sa précision de cette façon. Par contre, un programme instable qui contient par exemple des annulations dites catastrophiques demeure instable ; une augmentation de la précision ne se traduit pas nécessairement par une réduction de l'erreur (Goldberg, 1991).

Pour les cas dans lesquels la précision double n'est pas suffisante, il est possible d'utiliser une librairie comme MPFR (Fousse *et al.*, 2007), qui implémente une arithmétique de précision arbitraire et permet donc d'utiliser une précision supérieure. Elle étend les définitions de la norme IEEE à des précisions supérieures en garantissant un arrondi exact pour les opérations de base. Une librairie équivalente, CUMP, a récemment été implantée (Nakayama et Takahashi, 2011) et permet l'utilisation d'arithmétique de précision arbitraire sur GPU. L'avantage de cette méthode est qu'il y a peu de modifications à faire à un programme pour l'utiliser ; il suffit de changer les types des variables dont on veut augmenter la précision et d'initialiser la librairie. Par contre, lorsqu'un programme contient des instabilités, celles-ci ne sont pas nécessairement corrigées et leur source n'est pas indiquée. Par ailleurs, les calculs en précision supérieure à double prennent beaucoup plus de temps à s'exécuter.

2.3.2 Analyse manuelle

Lorsqu'une précision augmentée ne suffit pas à limiter l'erreur dans un programme, il est nécessaire de procéder à son analyse. L'ouvrage de Higham (2002) fournit une excellente introduction aux principes de l'analyse d'algorithmes exécutés avec des nombres à virgule flottante. De façon générale, sa méthode consiste à trouver une expression mathématique de l'erreur contenue dans un algorithme à partir des règles énoncées par la norme IEEE.

Deux types d'erreurs sont discutés dans ce livre. L'erreur en aval (*forward error*) est la différence entre le résultat obtenu par un calcul et le résultat réel – si le calcul était effectué avec une précision infinie. Cette erreur est généralement celle dont on cherche à calculer les bornes lors de l'analyse d'un programme. Une autre mesure utile est celle de l'erreur inverse (ou erreur en amont, *backward error*), qui consiste en la différence entre l'entrée du programme – déjà arrondie – qui donne un certain résultat en virgule flottante et l'entrée réelle exacte qui donnerait ce résultat. Un algorithme est considéré comme numériquement stable s'il produit toujours un résultat dont l'erreur inverse est faible.

Pour cette étude, les méthodes présentées par Higham sont intéressantes d'un point de vue théorique surtout, étant donné qu'elles se limitent à des algorithmes en algèbre linéaire, comme l'inversion de matrice et la résolution de systèmes d'équations. Différentes méthodes pour effectuer une série de sommes sont présentées, mais ne sont pas directement adaptables à `GPUMCD`, puisque l'accumulation des résultats se fait de façon parallèle et qu'il n'est pas possible de fixer un ordre dans lequel les éléments sont additionnés. Cependant, d'autres méthodes d'analyse sont discutées brièvement, comme l'analyse courante et l'analyse par perturbation, qui seront présentées plus loin.

Plusieurs auteurs ont appliqué cette méthode manuelle pour évaluer ou développer des algorithmes relativement courts. Castaldo *et al.* (2009) développe un algorithme qui calcule un produit scalaire entre deux vecteurs avec une erreur réduite par rapport aux algorithmes existants, sans toutefois augmenter le temps de calcul. Il utilise les résultats de Higham (2002) pour borner l'erreur sur les différents algorithmes étudiés. Les auteurs obtiennent une formule paramétrée qui donne des limites supérieure et inférieure à l'erreur produite par tous ces algorithmes, selon la taille et le nombre de subdivisions effectuées sur les vecteurs. Ils développent ensuite un algorithme qui utilise les paramètres minimisant la formule.

De son côté, Jiang et Stewart (2006) applique la méthode d'analyse en amont/aval au problème de l'enveloppe convexe – parmi un nuage de points, il s'agit de déterminer les points qui correspondent aux sommets d'un polygone dont la surface contient tous les autres points. Son objectif est de prouver que les nombres à virgule flottante sont appropriés pour les calculs en géométrie computationnelle. Il utilise en particulier l'erreur inverse pour déterminer à quel point une méthode approche la réalité. Un algorithme stable devrait ainsi pouvoir limiter le

problème d'imprécision dans les données en entrée, dans la mesure du possible. Selon la définition de stabilité par rapport à l'erreur inverse, un tel algorithme est considéré comme stable s'il calcule la solution exacte d'un problème (idéal) dont la différence avec le problème utilisé en entrée est inférieure à l'incertitude sur les données. En procédant à l'analyse d'un algorithme (connu pour être instable) pour résoudre le problème de l'enveloppe convexe, il découvre que l'instabilité provient d'un test dépendant d'un calcul de précision infinie, qui détermine si un point se situe à l'intérieur ou à l'extérieur de l'enveloppe. Ils expliquent qu'en adoptant plutôt un test plus approprié, cette dépendance sur une précision infinie est évitée. Un tel résultat comporte un certain intérêt pour `GPUMCD`, puisque son algorithme de traçage dépend justement du placement de particules dans le bon voxel et de l'intersection de rayons avec des surfaces tridimensionnelles.

Il existe par ailleurs des moyens de corriger des erreurs dynamiquement, lors de l'exécution d'un algorithme. Par exemple, Brisebarre *et al.* (2011) présente une méthode pour obtenir des bornes très serrées sur le résultat d'un calcul, à l'aide d'algorithmes de précision augmentée. Le principe de ces algorithmes est qu'ils retournent, en plus de leur résultat, l'erreur sur celui-ci, calculée au cours de leur exécution. Cette erreur est généralement beaucoup plus petite que le résultat et son incertitude est faible – ou même nulle, dans certains cas. Ils utilisent des fonctions de précision augmentée pour développer un algorithme qui calcule une racine carrée arrondie exactement, en faisant appel à l'opération FMA. Ils développent et analysent ensuite deux algorithmes qui calculent des normes de vecteurs en deux dimensions.

En résumé, l'analyse manuelle permet d'obtenir un aperçu de la stabilité ou de l'instabilité d'une méthode, et ainsi de confirmer sa justesse ou trouver un moyen de l'améliorer. Cependant, cette technique est sujette à beaucoup d'erreurs de la part de ceux qui la mettent en pratique et est fastidieuse à appliquer sur des programmes ou algorithmes plus complexes.

2.3.3 Analyse assistée

Pour résoudre les problèmes de l'analyse manuelle – le risque d'erreur et la difficulté de la mise à l'échelle – des techniques permettant une analyse assistée par ordinateur ont été développées. Elles nécessitent une représentation interne des nombres réels, des virgules flottantes, des erreurs qui leur sont associées ainsi que des opérations possibles. Les deux modèles les plus couramment utilisés sont l'arithmétique par intervalles et l'arithmétique affine.

Arithmétique par intervalles

Tel que décrit par Rokne (2001), l'arithmétique par intervalles utilise des bornes inférieure et supérieure pour représenter un nombre. Chaque opération sur des nombres donne lieu à un nouvel intervalle qui contient nécessairement le résultat exact. Cette représentation permet des calculs relativement rapides. Cependant, les intervalles peuvent s'agrandir de façon disproportionnée durant les calculs étant donné qu'ils ne contiennent aucune information sur les relations des erreurs entre les variables. Par exemple, l'opération $x - x$ donnerait comme résultat un intervalle deux fois plus grand que celui sur x, même s'il ne devrait contenir que zéro.

Arithmétique affine

Pour résoudre le problème de dépendance entre les variables dans l'arithmétique par intervalles, de Figueiredo et Stolfi (2004) développe l'arithmétique affine. Ce modèle représente une quantité x par l'expression

$$\hat{x} = x_0 + x_1\varepsilon_1 + ... + x_n\varepsilon_n$$

où les ε_i sont des symboles de bruit dont la valeur se trouve entre -1 et 1. x_0 constitue la valeur centrale de la variable et les x_i sont les déviations partielles associées avec les symboles de bruit. Cette représentation peut facilement être convertie en intervalle, et vice-versa. Chaque opération introduit un nouveau symbole de bruit dans le résultat, en plus de l'ensemble des symboles des facteurs de l'opération. Cependant, lorsque ces facteurs ont des symboles de bruit en commun – les coefficients correspondant à ces symboles sont différents de zéro – ceux-ci sont combinés dans le résultat. Cette méthode permet d'obtenir au final des bornes beaucoup plus serrées que l'arithmétique par intervalles. Par contre, elle nécessite un temps de calcul et une quantité de mémoire beaucoup plus élevés, qui peuvent augmenter exponentiellement avec le nombre d'opérations.

Applications

Harrison (2000) développe l'outil HOL Light pour vérifier le calcul des fonctions sinus et cosinus de l'architecture IA-64 en double précision étendue. Il vérifie toutes les étapes des calculs et tient compte des erreurs de troncature et d'arrondi. Le système HOL Light permet de générer des preuves mathématiques à l'aide d'une série d'axiomes – par exemple, un nombre qui comporte une erreur d'arrondi peut être représenté par $x(1+e)$, où $e \leq 1ulp$. En lui fournissant une description mathématique des opérations effectuées par un algorithme, il

peut déterminer entre autres la validité d'une sommation pour calculer une certaine quantité (pi, sinus, *etc.*) ou des bornes sur l'erreur causée par la troncature d'une telle sommation. Cet outil est utile pour valider un algorithme de façon assistée, mais il comporte toujours le risque d'erreur humaine.

Pour réduire les erreurs méthodologiques des preuves manuelles et rendre accessible à un plus grand public la génération de preuves mathématiques complexes, De Dinechin *et al.* (2011) propose l'outil Gappa. Celui-ci permet de certifier un programme avec virgules flottantes en automatisant l'évaluation et la propagation des erreurs d'arrondi tout au long du calcul et en vérifiant des propriétés particulières à l'aide d'assertions. Pour y parvenir, il effectue les calculs à l'interne à l'aide d'arithmétique par intervalles. Gappa identifie cependant les nombres représentables exactement, pour permettre de réduire les bornes sur l'erreur finale. Dans certains cas, il peut réécrire le calcul de l'erreur, selon certaines formules connues contenues dans une base de données, afin de trouver des bornes plus serrées. À la fin du calcul, il donne une preuve vérifiable de façon automatique de la validité des assertions et des bornes obtenues. Il est à noter que la validité de la preuve donnée ne dépend pas du bon fonctionnement de Gappa. L'inconvénient de ce logiciel est qu'il faut traduire les expressions impliquant des virgules flottantes dans le langage de Gappa, ce qui comporte encore un risque d'erreur, et qu'il ne fonctionne que sur des segments de code qui contiennent peu de branches, qui n'ont aucune boucle et dont la longueur est limitée.

Linderman *et al.* (2010) présente Gappa++, une extension de Gappa, qui ajoute plusieurs améliorations. L'objectif de ce nouvel outil est l'optimisation de programme plutôt que seulement la vérification, pour aider les développeurs de logiciel à découvrir des moyens d'augmenter la précision et la stabilité de leur programme. L'ajout le plus important en matière de précision des bornes calculées sur l'erreur est l'utilisation d'arithmétique affine plutôt que par intervalles. Cependant, l'aspect le plus intéressant pour l'étude présente est l'adaptation de règles spécifiques aux cartes graphiques NVIDIA, en ce qui concerne les arrondis, certaines fonctions mathématiques et l'opération FMA. Pour tester Gappa++, les auteurs ont notamment utilisé un script pour traduire directement les fichiers PTX (assembleur) produits par le compilateur de CUDA, ce qui élimine le risque d'erreur humaine dans la preuve générée.

Nguyen et Marché (2011) présente un outil similaire, qui cherche cette fois-ci à vérifier qu'aucune opération invalide (qui résulte en infini ou NaN, par exemple) n'est effectuée. L'outil analyse le code assembleur d'un programme pour générer des assertions et des obligations de preuve, qui sont ensuite prouvées automatiquement à l'aide d'un autre logiciel (comme Gappa). Le code assembleur est traduit dans le langage de Why (Filliâtre et Marché, 2007), dans lequel les auteurs ont modélisé les différents aspects d'un programme lors

de son exécution (types, opérations, registres, mémoire, etc.). Le principal intérêt de cette méthode est qu'elle tient compte non seulement du code source, mais de la plate-forme et du mode de compilation du programme analysé. Par contre, elle ne simule qu'un seul bloc d'instructions à la fois (aucune boucle ou instruction conditionnelle), ce qui en limite l'utilité pour un programme plus complexe.

2.3.4 Analyse statique

Interprétation abstraite

Pour effectuer une analyse plus complète de code source, plusieurs chercheurs font appel à l'interprétation abstraite. Cette théorie a été formalisée initialement par Cousot et Cousot (1977). Il s'agit de représenter un programme dans un domaine abstrait, plus général que l'ensemble des états concrets de ce programme, afin d'en déterminer certaines caractéristiques. Cette représentation peut servir entre autres à optimiser un programme, éliminer des parties qui ne seront jamais exécutées, détecter des bogues et, pour ce qui concerne cette étude, déterminer la fiabilité des calculs en virgule flottante.

Une des premières implémentations fonctionnelles en ce qui a trait spécifiquement à la représentation en virgule flottante est présentée par Goubault (2001). L'objectif de son outil est la détection automatique d'erreurs, c'est-à-dire des exceptions non gérées ou des résultats rendus imprécis ou invalides à cause d'erreurs d'arrondi. Son intention est également d'évaluer la précision des variables sans être trop pessimiste et permettre ainsi d'estimer la fiabilité des branches, lorsque les conditions impliquent des nombres à virgule flottante. Le domaine abstrait qu'il utilise est basé sur l'arithmétique par intervalles, pour laquelle il définit un langage simple sur les opérations en virgule flottante. L'inconvénient principal de cette représentation est qu'elle ne tient pas compte de la relation entre les variables, ce qui donne généralement une approximation grossière des valeurs possibles et résulte en un grand nombre de fausses alertes – indication d'une erreur là où il n'y en a pas.

Blanchet *et al.* (2003) démontre par contre que l'analyse statique par interprétation abstraite peut être à la fois efficace et fiable. L'objectif de l'outil développé est de découvrir les opérations indéfinies ou erronées – dépassement ou division par zéro, par exemple. La stratégie adoptée par les auteurs est d'améliorer progressivement un analyseur déjà existant en l'utilisant sur un programme considéré stable, jusqu'à ce qu'il n'y ait aucun faux résultat positif. Pour y parvenir, de nouveaux domaines abstraits adaptés à des situations spécifiques sont développés, qui permettent de tenir compte des relations entre variables et ainsi de réduire significativement les bornes trouvées à l'aide d'arithmétique par intervalles. Cet outil peut par ailleurs s'adapter à différents programmes analysés à l'aide de certains paramètres

– comme la profondeur maximale à laquelle développer les boucles ou séparer les différentes branches – et ainsi trouver les bornes les plus serrées possible. Cependant, il est conçu pour un type de programme très spécifique, généré par ordinateur pour des systèmes embarqués, avec des structures et des fonctions limitées ; il serait ainsi peu adapté à l'analyse d'un programme plus général.

Un outil similaire, appelé FLUCTUAT, est présenté par Goubault et Putot (2006) et Goubault *et al.* (2008). Son objectif est d'analyser des programmes industriels d'instrumentation et de contrôle et, pour un certain intervalle de valeurs d'entrée, donner l'intervalle des valeurs de sortie possibles. Il vise également à quantifier la perte de précision causée par les opérations en virgule flottante. Le domaine abstrait qu'il utilise est basé sur l'arithmétique affine, afin de conserver toutes les relations entre les variables. Les nombres réels sont donc représentés sous forme affine, avec des termes supplémentaires pour exprimer l'erreur induite par les nombres à virgule flottante. Ces derniers termes servent à calculer la différence maximale entre les résultats qui seraient obtenus avec des nombres réels – de précision infinie – et ceux obtenus par l'exécution du programme. Une difficulté importante de l'implémentation de FLUCTUAT est qu'il doit représenter les nombres réels qui définissent les coefficients de l'erreur à l'aide de nombres à virgule flottante. Les auteurs utilisent pour ce faire une précision supérieure à l'aide de la librairie MPFR. Un autre aspect particulier à considérer concerne la simulation des branches. En effet, pour limiter le temps de calcul, l'outil assume que les branches prises par les nombres réels et les nombres à virgule flottante sont les mêmes ; il émet cependant un avertissement lorsqu'elles pourraient être différentes.

Vérification de modèle et satisfaction de contraintes

Plusieurs chercheurs se sont basés sur FLUCTUAT pour développer des méthodes d'analyse statique, à l'aide d'interprétation abstraite et d'autres techniques. Ivancic *et al.* (2010), entre autres, a implémenté un outil, F-SOFT, pour détecter des calculs instables dans un logiciel. Cet outil génère un modèle inspiré des méthodes d'interprétation abstraite de FLUCTUAT, mais avec une arithmétique par intervalle. Ce modèle est ensuite vérifié de façon automatique, pour déterminer si un état erroné est atteignable. Cette méthode est cependant peu pratique pour de gros programmes étant donné que la taille du modèle et, par le fait même, le temps de vérification augmentent de façon exponentielle avec la longueur et la complexité du programme analysé.

Ponsini *et al.* (2012), de son côté, utilise directement FLUCTUAT dans l'outil présenté, rAiCp. Celui-ci améliore les bornes obtenues par FLUCTUAT en abordant le problème des relations entre variables d'un point de vue de satisfaction de contraintes. À chaque point d'exécution du programme analysé, les états possibles sont représentés par un en-

semble de contraintes sur les variables – chaque instruction ajoutant des contraintes. Ces contraintes sont résolues sur des intervalles dans le domaine des réels avec le programme RealPaver (Granvilliers et Benhamou, 2006) et dans le domaine des virgules flottantes avec FPCS (Michel, 2002). Le principal gain de rAiCp sur FLUCTUAT est effectué dans la résolution des branches. À chaque bloc conditionnel, l'analyse de FLUCTUAT est suspendue, des contraintes sont générées pour chaque branche et les résultats obtenus sont réunis lorsque les branches se rejoignent. Ainsi, l'outil obtient des résultats au moins aussi bons que ceux de FLUCTUAT, avec un temps d'exécution environ deux fois supérieur.

Le logiciel rAiCp est également comparé à CDFL, développé par D'Silva *et al.* (2012). Cet outil consiste en l'adaptation d'un solutionneur pour problèmes de satisfaction de contraintes pour traiter des intervalles de virgules flottantes. À l'aide d'interprétation abstraite, il crée un graphe du flot de contrôle (*control flow graph*, CFG) du programme analysé et associe un intervalle avec chaque variable et position dans ce graphe. Il génère ensuite un graphe de conflit abstrait (*abstract conflict graph*), avec un état initial – le début de l'exécution – et un état erreur, qui peut correspondre à une opération invalide ou un dépassement de certaines limites. Le solutionneur analyse toutes les traces possibles – des séries d'états reliés – et détermine qu'un programme est correct si aucune trace ne lie l'état initial à l'état erreur. Cet outil comporte le même inconvénient que les vérificateurs de modèle, c'est-à-dire un long temps d'exécution pour des programmes complexes. Cependant, il donne des résultats aussi bons que ceux de rAiCp, selon les tests effectués par Ponsini *et al.* (2012).

Autres outils

Finalement, d'autres travaux en interprétation abstraite ont mené à des outils légèrement différents. Dans Martel (2007), l'outil présenté mesure un indice de qualité – proportionnel à la différence entre la valeur réelle théorique et le résultat en virgule flottante – sur l'implémentation particulière d'une formule et en transforme les expressions en expressions équivalentes, mais plus stables numériquement. Chaque nombre est représenté dans le domaine abstrait par une paire de valeurs : une pour le nombre à virgule flottante et une pour l'erreur contenue dans ce nombre. Pour identifier précisément les points qui introduisent le plus d'erreurs, celle-ci est représentée par une série d'intervalles – un pour chaque point de contrôle du programme. Dans le domaine concret, cette erreur est tout simplement la somme des erreurs représentées.

Pour transformer les expressions, une règle d'équivalence est ajoutée à l'outil ; elle contient des règles pour différentes propriétés arithmétiques, comme l'associativité ou la distributivité. La profondeur à laquelle les expressions peuvent être transformées est limitée par un paramètre, afin de réduire le temps de calcul. Pour chaque transformation, l'outil calcule la

sémantique complète afin de déterminer la meilleure.

Martel (2012) présente aussi un outil similaire, RangeLab, qui calcule l'étendue possible des sorties d'un programme simple pour un intervalle d'entrées donné, ainsi qu'une borne sur l'erreur d'arrondi par rapport à un calcul de précision infinie. RangeLab utilise l'arithmétique par intervalles pour représenter les nombres à virgule flottante et leur erreur, de manière similaire à l'outil précédent, et l'interprétation abstraite pour gérer les branches. Comme avec les autres techniques d'interprétation abstraite, les branches et les boucles causent cependant une surapproximation des bornes. Pour améliorer ces résultats, RangeLab divise les intervalles afin de les analyser séparément et ainsi réduire l'erreur calculée. L'utilité de cet outil se limite cependant à des fonctions relativement simples, il ne peut pas servir à analyser des programmes complets.

2.3.5 Analyse dynamique

La principale difficulté des méthodes présentées précédemment est d'établir des bornes les plus serrées possible, soit sur les valeurs calculées par un programme, soit sur l'erreur dans le résultat. Ces méthodes doivent souvent traiter un intervalle sur les valeurs possibles en entrée et doivent donc rester assez générales pour garantir que les résultats obtenus soient corrects, c'est-à-dire sans faux négatif. À l'inverse, une méthode d'analyse dynamique calcule l'erreur en exécutant le programme avec des données spécifiques. Cela permet d'obtenir une mesure précise de l'erreur finale et de celle des valeurs intermédiaires pour cette exécution, mais sans garantir que cette erreur est valide dans tous les cas.

L'approche la plus simple – qui ne requière pas d'outil externe pour fonctionner – est l'analyse courante, mentionnée brièvement par Higham (2002). Elle consiste en l'ajout d'instructions pour calculer manuellement l'erreur sur les opérations en virgule flottante au fur et à mesure qu'elles se produisent. Elle nécessite cependant de savoir comment calculer cette erreur pour chaque opération et comporte le risque d'introduire de nouvelles erreurs. Plusieurs méthodes automatisées d'analyse existent cependant, basées sur le suivi des calculs durant l'exécution d'un programme.

Benz *et al.* (2012) présente une méthode qui donne un résultat similaire à l'augmentation de précision, mais sans modification au code source du programme analysé. Leur outil ajoute au programme exécuté, pour chaque variable, une valeur "fantôme" de précision supérieure, pour simuler sa valeur réelle. À chaque étape du programme, la différence entre une variable et son fantôme est calculée et une erreur est détectée lorsque cette différence dépasse un certain seuil. Les annulations "catastrophiques" – qui peuvent aussi affecter les nombres à virgule flottante de plus grande précision – sont également détectées, à l'aide d'un indicateur du danger d'une soustraction, qui compte le nombre de bits annulés par l'opération.

Cette méthode est en général moins efficace que les approches statiques, étant donné que les valeurs utilisées pour un test ne révèlent pas nécessairement les instabilités qui seraient présentes dans le programme. Elle ne permet pas non plus de déterminer la fiabilité des branches ; une telle fonctionnalité serait cependant relativement facile à implémenter. Un autre inconvénient majeur est que l'instrumentation du code exécutable augmente le temps de calcul par un facteur de 160 à plus de 1000. Par contre, elle permet de déterminer l'origine des instabilités dans le code, ce qui facilite grandement leur correction.

Arithmétique stochastique discrète

Vignes (2004) présente l'arithmétique stochastique discrète (*discrete stochastic arithmetic*, ou DSA), une théorie plus robuste sur laquelle baser une analyse dynamique. Cette théorie probabiliste modélise l'erreur contenue dans la représentation à virgule flottante d'un nombre par une variable aléatoire uniformément distribuée. L'erreur accumulée par une variable au cours de l'exécution d'un programme peut ainsi être modélisée par une distribution presque gaussienne, centrée sur la valeur réelle de cette variable.

Il est possible de simuler l'erreur uniformément distribuée dans un programme en arrondissant le résultat de chaque calcul aléatoirement vers le haut ou vers le bas. En exécutant le programme plusieurs fois, différentes valeurs sont obtenues pour les résultats. En tenant compte de la moyenne et de la variance de ces différentes valeurs, un test de Student permet d'estimer avec une certaine confiance le nombre de bits exacts dans le résultat.

Pour conserver la validité de cette méthode lors de l'analyse d'un programme instable, le concept de zéro informatique est introduit. Un résultat stochastique est considéré comme zéro dans ce sens si toutes ses valeurs sont nulles ou si le nombre de chiffres exacts est inférieur ou égal à zéro. En effectuant les exécutions multiples de façon synchrone, il est donc possible de vérifier en continu si le résultat d'une variable est toujours valide.

Jézéquel et Chesneaux (2008) implémente la DSA dans la librairie CADNA. L'utilisation de cette librairie (en Fortran, C ou C++) requiert quelques modifications au code source du programme à analyser : les types des variables à virgule flottante doivent être remplacés par les types dérivés définis par CADNA et la librairie doit être initialisée à l'aide d'une instruction. Les types dérivés contiennent en fait trois variables du type représenté. Les opérateurs mathématiques sont redéfinis pour ces types pour effectuer chaque calcul trois fois, avec un mode d'arrondi différent choisi aléatoirement chaque fois. Selon les options utilisées pour initialiser CADNA, une série de compteurs peuvent être incrémentés lorsque différentes conditions sont remplies, comme une annulation, un choix de branche incertain ou une valeur trop imprécise. Bien qu'elle permette de détecter de façon fiable toutes sortes de problèmes, cette méthode ne donne cependant pas d'indication sur la localisation de ces erreurs dans

un programme. En ce qui concerne l'impact sur le temps d'exécution, l'application examinée requiert environ trois fois plus de temps, ce qui est beaucoup plus rapide qu'avec d'autres techniques d'analyse dynamique.

Li *et al.* (2012) étend l'arithmétique stochastique discrète à des valeurs vectorielles. Son objectif est de pouvoir calculer des bornes serrées sur l'erreur d'algorithmes en algèbre linéaire, afin de déterminer s'il est possible d'utiliser une précision plus faible. Une réduction de la précision permet des implémentations beaucoup plus efficaces de différentes librairies sur des plates-formes parallèles, comme les GPU.

Li *et al.* (2011) étudie par ailleurs la robustesse de la DSA lorsque les hypothèses sur lesquelles elle est basée ne sont pas valides. Les résultats les plus importants sont que, d'après leurs tests, la modélisation d'une distribution non gaussienne de l'erreur sur un résultat ne cause pas une grande différence sur le nombre de chiffres exacts calculé – moins de 0.5 – et que les autres violations de ces hypothèses peuvent être détectées facilement lors de l'exécution de CADNA.

Les mêmes auteurs ont également développé une méthode pour appliquer l'arithmétique stochastique discrète sans modification du code source (Li *et al.*, 2012). Leur outil modifie plutôt le code assembleur pour introduire des instructions qui sélectionnent aléatoirement le mode d'arrondi. Pour les programmes sur CPU, un seul exemplaire du code doit être compilé, puisque le mode d'arrondi peut être changé durant l'exécution ; il suffit ensuite de l'exécuter plusieurs fois. Pour les programmes sur GPU, le mode d'arrondi doit être sélectionné à la compilation. Il faut donc créer plusieurs exemplaires du programme, dans lesquels les modes d'arrondi ont été préalablement choisis pour chaque opération en virgule flottante. Il suffit ensuite d'exécuter chacun de ces exemplaires et de collecter les différents résultats.

Le principal inconvénient de cette technique par rapport à celle décrite par Vignes est que les exécutions ne sont pas synchrones. Les variables ne peuvent donc pas être vérifiées au cours de l'exécution et aucune information n'est obtenue par rapport aux types d'erreurs présentes, s'il y en a.

Une méthode avec une composante similaire à la DSA a été développée par Tang *et al.* (2010). Elle consiste tout d'abord à perturber les différentes valeurs de nombres à virgule flottante à travers l'application, puis à perturber les expressions pour lesquelles des instabilités potentielles ont été détectées. Pour la perturbation de valeurs, plutôt que de simplement choisir le mode d'arrondi, ce sont les k derniers bits qui sont déterminés aléatoirement. Après un grand nombre d'exécutions de l'application ainsi modifiée, la différence maximale et le coefficient de variation (CV) entre les différentes valeurs obtenues sont calculés pour chaque variable. Un coefficient élevé indique que la variable varie beaucoup et qu'elle contient donc une grande erreur. Puisque cette manière de perturber les valeurs introduit une erreur

relativement grande, il est nécessaire de vérifier si une instabilité est effectivement présente en perturbant les expressions dont les variables ont un CV élevé. L'outil présenté génère ainsi différentes versions algébriquement équivalentes des expressions ciblées, selon la méthode décrite par Martel (2007). Si les résultats de l'application varient beaucoup pour diverses versions d'une expression, les auteurs considèrent qu'une instabilité y est présente.

2.4 Objectifs

L'objectif général de cette étude est de déterminer si des sources d'inexactitude sont présentes dans le code de `bGPUMCD`, d'en examiner l'impact et de généraliser ce travail en développant une méthode pour détecter ces sources.

2.4.1 Objectifs spécifiques

1. Quantifier l'erreur dans les calculs utilisant des nombres à virgule flottante de précision simple par rapport à double.
2. Quantifier l'erreur causée par les implémentations matérielles sur GPU – appelées intrinsèques – de certaines fonctions par rapport à leur implémentation logicielle.
3. Quantifier de façon relative l'erreur liée à la discrétisation en voxels du domaine sur lequel les calculs sont effectués.
4. Mettre au point une méthode pour détecter automatiquement les opérations spécifiques dans le code source – reliées aux éléments mentionnés dans les objectifs précédents – qui mènent à une erreur de précision dans le résultat final.

Un aspect important de cette recherche porte également sur l'impact des erreurs et des corrections qui sont apportées pour résoudre les problèmes détectés. L'effet d'une erreur à un certain point dans le logiciel se remarque sur le résultat final lorsque les opérations subséquentes préservent ou amplifient cette erreur, jusqu'à la fin de l'algorithme ou du programme. Ainsi, la correction de l'erreur est utile principalement si elle améliore la précision du résultat final. Dans le cas contraire, son impact sur le temps de calcul doit être pris en compte avant de l'appliquer.

Ainsi, le choix de deux méthodes différentes pour effectuer le même calcul est souvent basé sur un compromis entre le temps de calcul et la précision du résultat. Lors de la correction d'une inexactitude, il est donc nécessaire de décider si le gain de temps du calcul imprécis est plus important que le gain de précision du calcul plus lent. Cette question est particulièrement appropriée pour une méthode stochastique telle que celle utilisée par `bGPUMCD`. En effet, une erreur de précision qui affecterait de façon significative une particule pourrait demeurer invisible parmi les millions de particules simulées. Étant donné la nature aléatoire de ces simulations, il faut qu'une erreur soit plus élevée que l'incertitude inhérente à la méthode de Monte-Carlo pour être détectable.

2.5 Hypothèses

Trois causes générales d'erreurs ont été identifiées dans les objectifs : la précision – simple ou double – des opérations effectuées, l'utilisation de fonctions intrinsèques et la discrétisation en voxels.

H1. *Il est possible de quantifier l'apport individuel de ces trois causes d'erreur.*

D'après les informations présentées plus tôt sur les erreurs liées à la précision des calculs et sur la structure de `bGPUMCD`, certaines parties semblent plus susceptibles de contenir des erreurs. En particulier, l'accumulation de l'énergie dans chaque voxel introduit une petite erreur pour chaque ajout. Comme ces erreurs sont essentiellement aléatoires, elles devraient en général s'annuler.

H2. *L'erreur finale dans chaque voxel demeure inférieure à 1% de la valeur du voxel.*

Également, les erreurs – même très faibles – qui apparaissent dans les calculs liés à la géométrie et au traçage de rayon peuvent avoir un impact beaucoup plus grand lorsqu'elles attribuent une particule au mauvais voxel. De tels résultats catastrophiques devraient toutefois être tempérés par le grand nombre de particules simulées.

H3. *Dans un petit nombre de voxels, une erreur inférieure à 10% de la valeur de chacun peut être observée.*

Plusieurs méthodes automatiques de détection d'erreurs reliées à la précision des nombres à virgule flottante ont été présentées dans la section 2.3. Cependant, ces méthodes sont conçues pour des programmes exécutés sur CPU.

H4. *Il est possible de détecter de façon automatique et spécifique les erreurs de précision dans un programme sur GPU.*

CHAPITRE 3

PRÉSENTATION DE L'ARTICLE

Ce chapitre présente l'article du prochain chapitre et résume la méthodologie utilisée pour atteindre les objectifs spécifiques décrits plus tôt (section 2.4.1). Avant de décrire en détail les tests effectués sur `bGPUMCD`, le flot d'exécution du programme est décrit, afin de faciliter la compréhension du reste de ce texte, et les techniques utilisées pour comparer différents ensembles de résultats sont expliquées, étant donné qu'elles ont un intérêt pour toutes les autres étapes de la démarche. Ensuite, les modifications et tests nécessaires pour déterminer l'impact général des causes d'erreur sont décrits, ainsi qu'une division du programme en sections susceptibles de contenir des types d'erreur différents. Ces tests plus généraux sont suivis par l'introduction artificielle d'erreurs, aléatoires et systématiques, afin de déterminer avec plus de précision les opérations ou les parties de l'algorithme qui introduisent les plus grandes imprécisions. Les autres démarches entreprises pour ce travail sont décrites au chapitre 5.

3.1 Fonctionnement de `bGPUMCD`

Initialisation et fin

Au début de l'exécution, plusieurs étapes servent à mettre en place les éléments nécessaires au déroulement de la simulation par le processeur graphique. Ces étapes sont majoritairement exécutées par le processeur central. Elles consistent en la lecture des options sélectionnées par l'utilisateur, la lecture des données décrivant le problème à simuler, leur transfert sur le processeur graphique et l'initialisation du générateur de nombres aléatoires.

Les données à lire incluent une description de la géométrie du volume simulé, qui peut provenir d'un tomodensitogramme ou d'un fichier fourni par l'utilisateur, les propriétés physiques des différents matériaux retrouvés dans ce volume et le type de source de radiation à simuler, ainsi que leur disposition à l'intérieur du volume.

Une fois que toutes les données sont en place sur la carte graphique, les particules sont générées au sein des sources retrouvées dans le volume et le noyau responsable de la simulation même – la boucle principale de `bGPUMCD` – est lancé. Étant donné que le nombre de particules simulées par un noyau est limité, ces deux dernières étapes – la génération des particules et l'exécution de la boucle principale – sont répétées jusqu'à ce que le nombre voulu de particules soit simulé. Tous les noyaux ajoutent leurs résultats, soit une certaine quantité de radiation dans chaque voxel, dans le même tableau. Une fois la simulation terminée, ce tableau, qui

contient la distribution de dose résultante, est copié vers l'hôte pour être écrit dans un fichier.

Boucle principale

Le coeur de la simulation consiste en le calcul du parcours de chaque particule, de ses interactions et de la quantité d'énergie qu'elle dépose dans chaque voxel traversé. Ce calcul est fait de façon parallèle, une particule par fil d'exécution CUDA.

Le parcours d'une particule est simulé par une boucle – appelée ici "boucle principale". Chaque itération de la boucle peut être divisée en trois étapes : calcul de la distance à parcourir lors du prochain "pas", dépôt d'énergie dans tous les voxels traversés lors de ce pas et simulation d'une interaction.

Le calcul de la longueur d'un pas est effectué par la fonction `calculePas`. Le libre parcours de la particule à travers le volume voxélisé est d'abord sélectionné, de façon aléatoire. Ensuite, pour tenir compte des différents matériaux qui composent les sources de radiation, ce libre parcours est modulé en fonction de la longueur du trajet de la particule dans chacun de ces matériaux. Ces différentes longueurs sont calculées par la fonction `calculeIntersection`, nommée `computeIntersection` dans l'article du prochain chapitre, où elle sera décrite plus en détail.

Ensuite, la quantité d'énergie déposée dans chaque voxel le long du pas de la particule est calculée. Elle dépend du matériau composant le voxel et de la longueur du segment de la trajectoire qui traverse ce voxel, calculée par la fonction `calculeSegment`.

Enfin, l'interaction qui se produit à l'extrémité du pas est sélectionnée et ses effets sont appliqués à la particule. Quatre interactions sont simulées par `bGPUMCD`. La première, l'effet photoélectrique, est une absorption de l'énergie d'un photon par un électron – non simulé par `bGPUMCD`. Le résultat de cette interaction est une perte d'énergie du photon, déterminée par la nature du matériau où il se trouve, et un changement de direction, déterminé aléatoirement. La deuxième interaction, la diffusion Compton, est une diffusion d'un photon sur un électron. Elle résulte en une déviation du photon et une perte d'énergie proportionnelle au changement de direction. La troisième interaction, la diffusion Rayleigh, est une diffusion élastique d'un photon par une petite particule. Elle résulte uniquement en un changement de direction, qui dépend du matériau qui diffuse le photon. La dernière interaction est en réalité une interaction fictive, qui laisse la direction et l'énergie de la particule inchangées. Elle est une composante importante de l'algorithme de Woodcock, qui sert à faciliter le transport des particules (Carter *et al.*, 1972). Cet algorithme considère le milieu où se déplacent les particules homogène, d'une densité égale à celle du matériau le plus dense qui s'y trouve. Lorsqu'une particule traverse des matériaux moins denses, il est possible que l'interaction sélectionnée soit l'interaction fictive, qui simule la probabilité moins élevée d'une interaction

dans ces matériaux.

Une fois que l'interaction est simulée, la boucle vérifie que la particule se trouve toujours dans le volume simulé et que son énergie est suffisamment élevée – au-delà d'un niveau déterminé par l'utilisateur. Si c'est le cas, un prochain pas est calculé pour cette particule ; sinon, la boucle se termine. Le noyau complet se termine lorsque toute ses particules arrivent à la fin de la boucle principale.

3.2 Quantification de l'erreur

Le résultat d'une simulation par `bGPUMCD` est un histogramme en trois dimensions, ou distribution de dose, dans lequel chaque emplacement contient la dose de radiation reçue par le voxel correspondant. Pour comparer deux distributions, il est nécessaire de tenir compte à la fois de la quantité de radiation et de sa répartition. Lors des comparaisons, une des deux distributions est considérée comme la référence ; l'autre distribution est appelée "évaluée" tout au long de ce travail. Deux méthodes sont utilisées pour effectuer les comparaisons au cours de ce travail et sont présentées ici.

3.2.1 Différence de dose

Les deux distributions sont comparées voxel par voxel. Pour chaque voxel de la distribution référence, la valeur absolue de la différence avec le voxel correspondant de la distribution évaluée est calculée. Ces différences sont additionnées, puis le résultat est divisé par la somme des valeurs de tous les voxels de la distribution de référence selon l'équation suivante :

$$\frac{\sum_{v \in V} |v_b - v_a|}{\sum_{v \in V} v_a} \times 100 \tag{3.1}$$

où v correspond à la dose à une certaine position de voxel, tandis que v_a et v_b représentent les voxels des deux distributions comparés (a étant la distribution référence). Le nombre obtenu indique la taille de la différence entre les deux distributions par rapport à la quantité totale de radiation déposée dans le volume de la distribution référence.

3.2.2 Histogramme des différences relatives

Un des inconvénients de l'indicateur présenté ci-dessus est qu'il n'offre aucune information sur la répartition de l'erreur totale. Par exemple, il est possible qu'une grande différence de dose entre deux distributions soit très localisée, générant ainsi un faible résultat pour la différence totale. La solution utilisée pour détecter une telle situation est de construire un histogramme de l'amplitude des erreurs, voxel par voxel. Ainsi, il est possible d'observer si la

différence globale est causée par une erreur régulièrement répartie à travers tout le volume ou si elle provient uniquement d'un petit groupe de voxels.

Il est également possible de présenter les différences relatives individuelles, sous forme de carte thermique. Cette représentation permet de visualiser l'emplacement de ces différences.

3.3 Contenu de l'article

3.3.1 Augmentation de la précision

Tous les nombres à virgule flottante de simple précision (*float*) sont remplacés par des nombre de double précision (*double*) et les résultats ainsi obtenus sont comparés aux résultats originaux. Le résultat des calculs en double précision est considéré plus près de la réalité.

Ces deux représentations sont étudiées pour différents aspects du programme. Les deux premiers aspects sont bien ciblés et sont susceptibles de contenir des erreurs significatives et de sources différentes ; le troisième regroupe simplement le reste du programme.

1. Accumulation de l'énergie par voxel
   Au cours de l'exécution, chaque particule ajoute de l'énergie aux voxels qu'elle traverse. Ainsi, la dose calculée dans chaque voxel est le résultat d'une sommation d'un grand nombre de petites quantités, qui peut atteindre plusieurs millions pour les voxels près d'une source de radiation. Plus le résultat augmente, plus les petites quantités qui y sont ajoutées perdent de la précision.

2. Traçage de rayon
   Le programme utilise les position et direction exactes de chaque particule pour déterminer la quantité d'énergie déposée au cours de son trajet, le voxel où cette énergie est déposée ainsi que la probabilité d'interaction – qui dépend de la densité du milieu, donc du voxel où se trouve la particule. Une légère imprécision pourrait placer une particule dans le mauvais voxel, ce qui introduirait une erreur significative dans les deux voxels impliqués. Elle pourrait également résulter en une interaction ou une absence d'interaction, qui causerait une erreur pour tous les dépôts de dose subséquents de la particule, ainsi que pour tous les dépôts qui auraient dû avoir lieu autrement.

3. Autres opérations
   Le reste des opérations sont reliées à des calculs de quantités physiques comme l'énergie des particules et la densité du milieu. Elles servent principalement à déterminer les interactions qui se produisent et leur effet. L'ensemble de ces calculs peuvent contenir des erreurs imprévues et seront testés séparément des deux autres sections décrites.

3.3.2 Fonctions intrinsèques

Pour les fonctions dont il existe une implémentation en matériel, remplacer l'appel par cette version et comparer les résultats. L'implémentation logicielle est considérée plus près de la réalité.

3.3.3 Tracé de rayon

La fonction `calculeIntersection` (`computeIntersection` dans l'article), utilisée pour calculer la distance parcourue par une particule dans un milieu, est analysée plus en profondeur, en raison des problèmes qu'elle cause et pour déterminer exactement leur origine.

3.3.4 Injection d'erreur

Des erreurs sont introduites à diverses parties du programme, afin de simuler des erreurs numériques importantes, pour déterminer leur impact sur l'ensemble du programme.

CHAPITRE 4

ARTICLE 1 : A STUDY OF POTENTIAL NUMERICAL PITFALLS IN GPU-BASED MONTE CARLO DOSE CALCULATION

Vincent Magnoux[1], Benoît Ozell[1], Sami Hissoiny[2], Éric Bonenfant[3,4] and Philippe Després[3,4]

[1] Département de de génie informatique et génie logiciel, École Polytechnique de Montréal, Montréal QC, Canada

[2] Elekta ltd.

[3] Département de radio-oncologie and Centre de recherche du CHU de Québec, Québec QC, Canada

[4] Département de physique, de génie physique et d'optique and Centre de recherche sur le cancer, Université Laval, Québec QC, Canada

Abstract

The purpose of this study was to evaluate the impact of numerical errors caused by the floating point representation of real numbers in a GPU-based Monte Carlo code used for dose calculation in radiation oncology, and to identify situations where this type of error arises. The program used as a benchmark was bGPUMCD. Four tests were performed on the code, which was divided into three functional components : energy accumulation, particle tracking and physical interactions. First, the impact of single-precision calculations was assessed for each functional component. Second, a GPU-specific compilation option that reduces execution time as well as precision was examined. Third, a specific function used for tracking and potentially more sensitive to precision errors was tested by comparing it to a very high-precision implementation. Fourth, errors were injected manually in various sections of the program to simulate large numerical errors.

Numerical errors were found in two components of the program. Because of the energy accumulation process, a few voxels surrounding a radiation source end up with a lower computed dose than they should. The tracking system contained a series of operations that abnormally amplify rounding errors in some situations. This resulted in some rare instances (less than 0.1%) of computed distances that are exceedingly far from what they should have been. Error injection testing suggested that no other numerical error in the code studied was large enough to affect simulation results. Most errors detected had no significant effects on the result of a simulation due to its random nature, either because they cancel each other out or because they only affect a small fraction of particles. The results of this work can be

extended to other types of GPU-based programs and be used as guidelines to avoid numerical errors on the GPU computing platform.

Keywords : Numerical analysis, floating-point precision, Monte Carlo methods, radiation therapy, parallel processing

Submitted to : *Physics in Medicine and Biology*, 18 July 2014

4.1 Introduction

Radiation therapy consists in using particles (photons, electrons or protons, for example) of sufficient energy to ionize/damage cell components with the objective of killing these cells. In order to deliver radiation doses that are recognized as having the optimal biological effect, an accurate knowledge of dose distributions resulting from a given irradiation setup is essential. The problem of dose deposition in a patient is most of the time analytically intractable because of the complex geometry involved. In the clinic, many approximations are made to approach a solution to the dose deposition problem. These approximations however can lead to inaccurate solutions. Monte Carlo techniques can also be used for dose calculation in radiation oncology. They are considered the most accurate as they are based on fundamental physics principles (energy-dependent interaction cross-sections, material properties). They consist essentially of simulating incident particles one by one and scoring voxel by voxel the energy deposited by each primary and secondary particle in the tissues. The dose is obtained by dividing the total energy received in a voxel by its mass. A random number generator is used in Monte Carlo techniques to determine the fate of a particle based on interaction cross-section data.

Multiple tools based on Monte Carlo techniques, such as EGSnrc (Kawrakow, 2000), PENELOPE (Baró *et al.*, 1995) and Geant4 (Agostinelli *et al.*, 2003), have been developed for the purpose of radiation transport simulation. These codes however are notoriously slow and long execution times have historically prevented the deployment of Monte Carlo solutions in regular clinical practice. In order to achieve shorter execution times, codes such as `GPUMCD` (Hissoiny *et al.*, 2011a) and gDPM (Jia *et al.*, 2011) have been implemented on a GPU platform. Their accuracy has been verified by comparing their results to physical measurements (Faddegon *et al.*, 2009) or to other validated codes (Kawrakow et Fippel, 2000; Jia *et al.*, 2011; Hissoiny *et al.*, 2011b). Most improvement efforts are typically focused on variance reduction techniques and on the accuracy of the underlying physical models, but do not explore the possibility of inaccuracies caused by the way the hardware platform performs computations, in particular on real numbers.

This study will examine floating point-related issues that arise in the context of a program executed on the GPU. The program used as a test case is `bGPUMCD` (Hissoiny *et al.*, 2012), an adaptation of `GPUMCD` (Hissoiny *et al.*, 2011b) for brachytherapy. The code implements a Monte Carlo method for brachytherapy dose calculations, using only photon transport in a mixed geometry. The geometry in `bGPUMCD` relies on a voxelized representation to account for patient-specific data (Computed Tomography images) and a parameterized representation to accurately model the radiation sources (small seeds containing ^{125}I). Two effects are expected

to oppose each other in a Monte Carlo simulation. On the one hand, the repetitive nature of the computations – a large number of photons are simulated – means that a small error in an operation could occur multiple times and have a large impact on the results. On the other hand, the random nature of the simulation could generate errors that cancel each other out.

This study focuses on finding numerical errors in `bGPUMCD`, in assessing their impact on the results and in verifying that no other numerical errors affect the program. The paper addresses aspects that are specific to GPUs or to `bGPUMCD`, but its conclusions hold for more general programs. Section 4.2 presents a background on floating point representation and numerical errors, section 4.3 describes the methods used here to detect these errors and section 4.4 discusses the test results and program changes necessary to correct the errors found.

`bGPUMCD` was not compared to other Monte Carlo codes here because the objective is to find errors rather than to improve the physical models or the algorithms used to describe them.

4.2 Background

4.2.1 Error generation and propagation

The use of a finite number of bits to represent real numbers requires a discretization process. This is most commonly accomplished by using a floating point representation. Floating points on NVIDIA GPUs follow the IEEE-754 norm in which 1 bit is used to specify the sign, 8 are used for the exponent and the 23 others are used to for the coefficient – in single precision (IEEE, 2008). These last bits are the ones that limit the precision during computations.

One measure of the error contained in a floating point number or between two numbers, which will be used throughout this paper, is the relative error. It is defined as $|(x - \hat{x})/x|$, where x is the ideal real or the reference floating point number and $\hat{x}$ is the compared floating point value.

As a result of the approximation of real numbers, errors can arise in most floating point computations from three different sources (Higham, 2002). First, the representation itself introduces an error if the number cannot be represented exactly in binary floating point form, for example π or even decimal 0.1. Second, any floating point operation may also introduce an error when its result is rounded. Finally, errors can come from the truncation of a formula : some operations may be implemented using Taylor series, in which case only the first few terms are used, thereby introducing an approximation error in the result.

In most cases, floating point errors have little impact on a program's results. They only

become a problem when they grow large enough to result in an output different from the theoretical one. The main mechanism by which errors can reach such a size is called *cancellation* (Goldberg, 1991; Higham, 2002). It occurs when two very close numbers containing some error are subtracted. Their most significant digits cancel each other out and the remaining part is left with a greatly amplified error. If the resulting number is then used in other computations, it can cause the program to output a wrong final result especially if the execution flow depends on it.

Another consequence of the floating point approximation of real numbers is that the usual arithmetic properties, like associativity and commutativity, no longer hold. This implies that two mathematically equivalent formulas will not necessarily yield the same result when executed on a computer. It also means that cancellation-related errors can sometimes be avoided by rearranging the terms of an equation to a mathematically equivalent one in which the potential cancellation is absent.

4.2.2 Error detection and correction

The simplest approach to correct errors caused by the floating point representation is to increase the precision of computations. Although it cannot completely eliminate errors, this method reduces their magnitude and thus their impact on the results of a program. However, simply increasing the precision of calculation does not reveal where errors occur in a program.

Automatic analysis methods can provide more insight into error propagation within a program. They can be divided into two broad categories. The first one, static analysis, is used to compute a range of real values that can result from floating point operations in a source code (De Dinechin *et al.*, 2011; Goubault *et al.*, 2008; Ponsini *et al.*, 2012; Martel, 2012). It can identify sequences of unstable operations, which can be defined as any operation resulting in a too wide range of values. The downside is that the ranges found are generally very large and thus lead to a high number of false positives.

The second category, dynamic analysis, consists of executing a program multiple times while either altering the results of each floating point operation or computing the exact error on these results (Benz *et al.*, 2012; Jézéquel et Chesneaux, 2008; Lam *et al.*, 2012; Tang *et al.*, 2010). An instability is detected when multiple executions do not lead to the same final result or when the error on a certain intermediate result is too high. This technique leads to more realistic results than static analysis since it is based on empiric observations of the running program, but it can also miss some errors.

In this work, tests related increasing precision and to dynamic analysis were performed as well as others which were conducted manually on more problematic components of the code.

4.3 Methods

Four sets of tests have been performed in order to check for numerical errors in `bGPUMCD`. The first two consist of modifying the precision of floating point operations to detect errors. The precision can readily be increased by using double precision, since all floating point computations in `bGPUMCD` are done in single precision by default. The precision can be decreased for some operations by using the `-use_fast_math` compilation option, which will be described later.

The third set of tests consists in a manual analysis of a specific, precision-sensitive function, accompanied by a comparison with a high-precision version of that function. This analysis is done by examining the sequence of operations and evaluating the risk of precision loss in each element of this sequence. Since this kind of analysis is more time-consuming, it was performed on a component of the program already shown as problematic.

The last set of tests is based on the same principles as dynamic analysis techniques. The source code was modified to inject errors in various parts of the program to simulate the effect of large numerical errors. While automatic analysis techniques are more systematic than this method, they are not well adapted for the program discussed here. The highly parallel nature of the GPU platform introduces some challenges – mainly caused by the ordering of computations – not taken into account by existing automatic analysis approaches. Also, the very repetitive computations conducted here mean that the objective is to find the frequency of errors rather than whether they occur. Most of the existing approaches except one (Li *et al.*, 2011) have not been tested on GPU programs. For the sake of simplicity, a manual method was preferred here.

The Monte Carlo code was divided into three functional components for the purpose of analysis : energy accumulation, particle tracking and physical interactions. The first component – energy accumulation – is the section of the code that simply accounts for energy deposited in a voxel by a traveling particle. This energy is simply added to the total energy count of that particular voxel. The second component of the code which could contain precision-related errors is the particle tracking system. Keeping track of the exact position of each particle allows the simulator to determine the likelihood of an interaction based on the material in which the particle is located. It also allows for the compilation of energy contributions to the appropriate voxel. The last component includes floating point operations in the program that are related to physical interactions between particles and the medium. It contains operations closely related to particle tracking but which do not directly displace the particles. This component also includes operations related to physical input data and to the random number generator at the core of the Monte Carlo simulation.

Two different simulation scenarios were tested here, both involving a 30 cm water cube divided into 1 mm^3 voxels. In the first case, a single radiation source was located at the center of the cube, leading to a simulation where approximately 60% of execution time is dedicated to the energy accumulation component of the program. In the second test case, 58 sources were positioned in the cube in a fashion similar to a real patient case. This scenario led to over 75% of execution time dedicated to the photon tracking system of `bGPUMCD`. A more accurate assessment of these proportions is made difficult by the fact that all functions are inlined, for a much shorter execution time, which prevents a separate evaluation by the CUDA profiler. In both cases, a total of 2 billion photons were simulated.

For every comparison between two dose distributions presented in this work, both distributions were generated by a simulation with the same seed for the random number generator. This removes any factor related to the stochastic aspect of the Monte Carlo method to concentrate only on precision-related errors or differences. All comparisons compute the voxel-per-voxel relative difference between the two distributions. When presenting difference maps, the sign of these differences is taken into account, but when presenting histograms of the distribution of the size of these differences, only their magnitude is considered.

4.3.1 Double-precision operations

Since execution speed is an important goal of `bGPUMCD`, all floating point operations are performed in single precision. Single-precision operations are typically twice as fast as their double-precision versions on GPUs, although the difference can be much larger on old hardware. The objective of the first test is to determine whether this precision is sufficient and, if not, to find a software solution that requires the minimal amount of double-precision operations in order to reach accuracy targets while limiting the impact on execution time. For this purpose, two components of `bGPUMCD` have been targeted as more likely to contain errors caused by the floating point representation : energy accumulation and particle tracking.

Energy accumulation results in a long summation which can represent millions of operations. When two numbers of a different order of magnitude are added, the lower bits of the smallest number are lost because they go beyond the floating point precision. A larger difference between the two numbers results in a greater loss of precision. As the energy in a voxel increases, so does the difference between the total and each contribution. This phenomenon makes each new addition less precise and could cause the final result in a voxel to be different from what it should have been, especially if it contains many individual contributions. However, the effect could be mitigated by the fact that rounding errors should be random and therefore cancel each other out.

The second critical component is particle tracking. Errors in particle location translate

into wrong energy depositions and also affects the occurrence of an interaction based on wrong material properties, which in turn can also lead to incorrect energy deposition afterwards. As with the energy accumulation process, the random nature of the simulation and the high number of particles could reduce the impact of such events.

Other floating point operations in the program are related to particle interactions, which mainly depend on the tracking system, on the physical data and on the random number generator. This component of the program was tested separately, although no major errors were expected other than those related to the tracking system.

In order to broadly evaluate the possible occurrence of numerical errors, all floating point computations present in `bGPUMCD` were converted to double-precision operations, one component at a time. For each of the three program components described above – energy accumulation, tracking system and physics – floating point operations were performed in single precision while leaving the other two components in double precision. The results – dose distributions – were compared to those obtained by the program running with all floating point computations done in double precision. Any difference can be attributed to the fact that one component of the program has been computed less accurately and thus reveal the presence of an inaccuracy in the single-precision version of that component.

4.3.2 Fast math option

CUDA provides mathematical functions similar to those found in the C math library, but in some cases, it also provides a hardware implementation of these functions, called intrinsic. Using CUDA's intrinsic functions greatly improves computation times with respect to the library implementation, since these functions map to a single hardware instruction. However, their results are slightly less accurate than those from the library functions, usually in some specific input range (NVIDIA, 2012), and they can potentially introduce some errors. In `bGPUMCD`, intrinsic functions can be used for trigonometric functions, square roots and division operations – in the last case with a faster, non IEEE-compliant algorithm.

The second test is thus aimed at determining whether using intrinsic functions has a noticeable effect on simulation results. Intrinsic functions were enabled globally in `bGPUMCD` by compiling it with the `-use_fast_math` option. Results obtained this way were compared to the results obtained without using intrinsics. Both scenarios described earlier were tested, using only single-precision operations since intrinsic functions are not all available in double precision.

4.3.3 Tracking system

Early testing revealed that a fault in the tracking system caused a major increase in computation times under certain circumstances. More specifically, the code responsible for computing distances in the parameterized representation of the geometry would output inaccurate results. The third test presented in this paper is related to this part of the code and was performed with two different approaches. The first one is a manual analysis of the function responsible for finding the intersection between a particle track and a quadric surface, called `computeIntersection`. The second one is a more systematic testing of this function by analyzing the accuracy of its output.

Surfaces are described algebraically in the parameterized representation of the geometry in `bGPUMCD` and are used to define source volumes for instance. When computing the distance between two interactions for a particle, the simulator must compute the distance between the particle and each surface in the whole simulation volume in order to determine the material through which the particle passes along its trajectory.

In the test scenarios described above, the only parameterized volumes are those representing brachytherapy seeds. The seeds are described using only spheres and cylinders, with a simplified quadric equation : $ax + by + cz + j = 0$, where a, b and c are scaling factors along each axis and $-j$ is the square of the radius of the sphere or cylinder. Algorithm 4.1 presents the pseudocode of `computeIntersection`. The algorithm works by parameterizing the particle trajectory, inserting that parameterization in the equation of the quadric surface and finding the roots t_1 and t_2 of that new equation.

Algorithm 4.1 was implemented as a separate program using the mpmath arbitrary-precision floating-point arithmetic library (Johansson *et al.*, 2013). When running the second test scenario described above (with 58 source seeds), input data to `computeIntersection` were randomly sampled, along with their corresponding output, and fed into the higher-precision program with the precision set to 100 decimals. The differences between the two sets of results reflect the magnitude of the error introduced when computations are performed in single precision. The errors contained in the results of 5 million calls to `computeIntersection` were recorded in order to give a general idea of their distribution. In reality, this function is called billions of times when simulating the second test case. Additionally, the double-precision version of `computeIntersection` was tested in the same way, to determine to what extent errors would be reduced by performing the computations at this level of precision.

ALGORITHM 4.1 The `computeIntersection` algorithm. p and d are 3D vectors containing the position and direction of the input particle ; M is a 3x3 matrix containing parameters a, b and c of the surface ; j is the last parameter in the quadric equation (see section 4.3.3), a negative value with magnitude equal to the square of the radius of the surface volume.

$A \leftarrow M \cdot d \cdot d$
$B \leftarrow M \cdot p \cdot d$
$C \leftarrow M \cdot p \cdot p + j$
$radical \leftarrow B^2 - 4AC$
if $radical < 0$ **then**
    return ∞
end if
$t_1 \leftarrow \frac{-B+\sqrt{radical}}{2A}$
$t_2 \leftarrow \frac{-B-\sqrt{radical}}{2A}$
if $t_1 > 0$ **then**
    return t_1
else if $t_2 > 0$ **then**
    return t_2
else
    return ∞
end if

4.3.4 Error injection

The fourth set of tests consisted in introducing errors at various points in the program and comparing the results of simulations with and without errors. The physical data and the results of various functions were altered, as they would be if the representational error or the accumulated rounding error were larger. The artificial errors were added at one place at a time and multiple error sizes were tested every time. This method could determine the size above which an error becomes detectable in the final results, which is when the difference between the error-injected result and the error-free result is higher than between two error-free executions of the program. This also determined which parts of the program are more sensitive to numerical errors and to what extent these errors cancel each other out in a Monte Carlo simulation.

In the physical data, errors were injected before doing any actual computations, when reading the data from files. The last n bits of each number were changed to simulate an increase in the representational error of these numbers, which is normally inferior to the value of the last bit (for a maximal relative error of 2^{-23}). For the other injected errors – on photon position, direction and on the return value of various functions – voxel size was used as a base unit for error size rather than a relative error, so that they all had a common

reference. This simulates the error accumulated through all computations.

In addition, geometric functions computing trajectory lengths in individual voxels and direction changes were also implemented in higher precision (100 decimal places), the same way as for `computeIntersection`, so that the error in their return value could be compared to the smallest error that leads to a detectable change in simulation results.

4.4 Results

4.4.1 Double-precision operations

Differences between single-precision and double-precision computations for the energy accumulation and tracking components of the program are showcased in figure 4.1 and 4.2 using relative difference maps. These figures only show a $60 \times 60 \times 60$ mm (or voxels) subset of the simulated volume to make the differences more visible.

Figure 4.1 shows that for most of the volume the results are identical but that there is an important difference between the two precisions for the energy accumulation component in the voxels close to the source, for the first test scenario. In this image, the difference reaches over 40% of the double-precision dose, in the voxel closest to the center of the source, but it is even higher in the ones partially occupied by the source. For brachytherapy applications, dose to the source itself has no interest. However, errors in voxels close to the source can have a significant impact. This difference can also be observed in the 58-source test case, but to a much lesser extent.

During a simulation, as the total energy deposited in a voxel grows, it can reach a point where subsequent contributions become too small : the single-precision addition operation is not precise enough to take them into account. This happens when individual contributions are approximately 2^{24} (or $\approx$16 million) times smaller than the total energy accumulated in a given voxel. This phenomenon is the same as the one described in section 4.3.1, but instead of a gradual loss of precision with each energy deposit, it causes a complete loss of some of these deposits. In the second test case, the simulated photons were distributed among all 58 sources so the effect around each of them was less severe, as the 2 billion simulated photons are spread across all sources. In other words, the accumulated energy in voxels do not reach the critical threshold as quickly as in the case of a single source.

A simple solution for this issue – other than using double precision – is to use multiple energy accumulation buffers to record energy deposition and to sum them after the simulation is completed. The number of energy accumulation buffers could be derived empirically from a test simulation and will depend on characteristics of the material involved and on the energy of primary photons. By staging the energy accumulation in multiple steps, the overly large

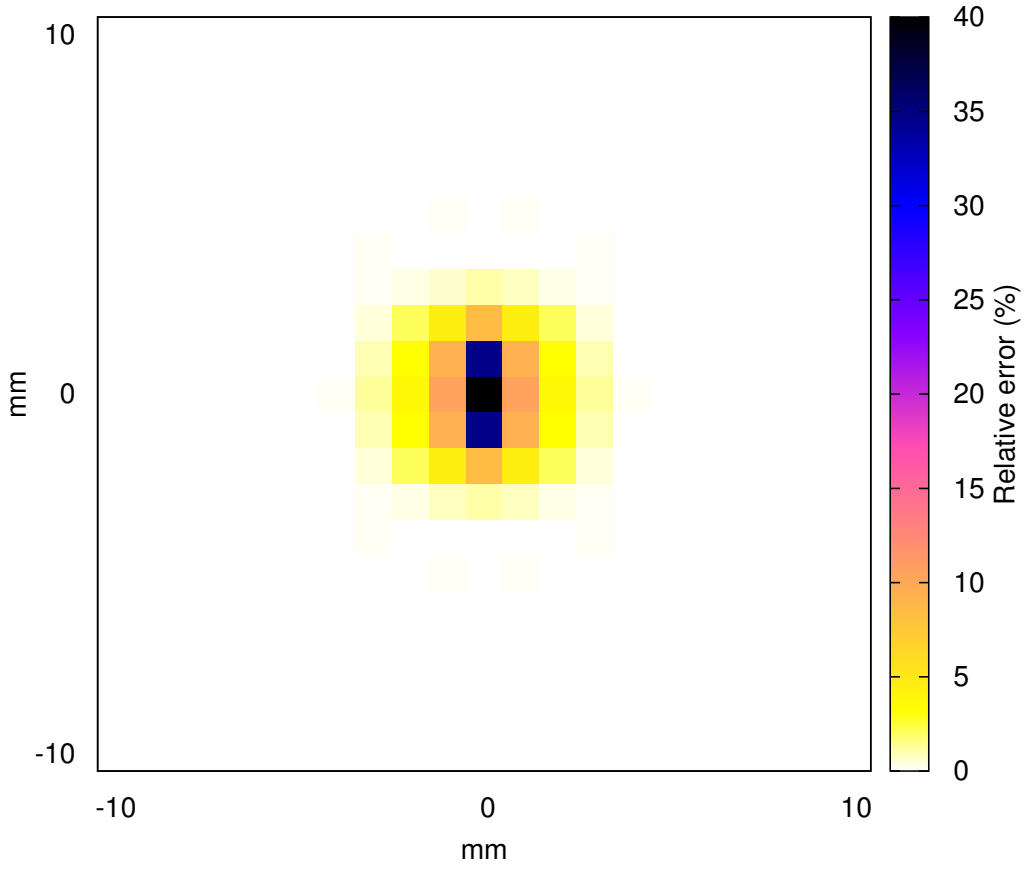

Figure 4.1 Voxel-per-voxel relative difference (in %) between single- and double-precision energy accumulation, in a plane adjacent to the radiation seed at the origin (test case 1).

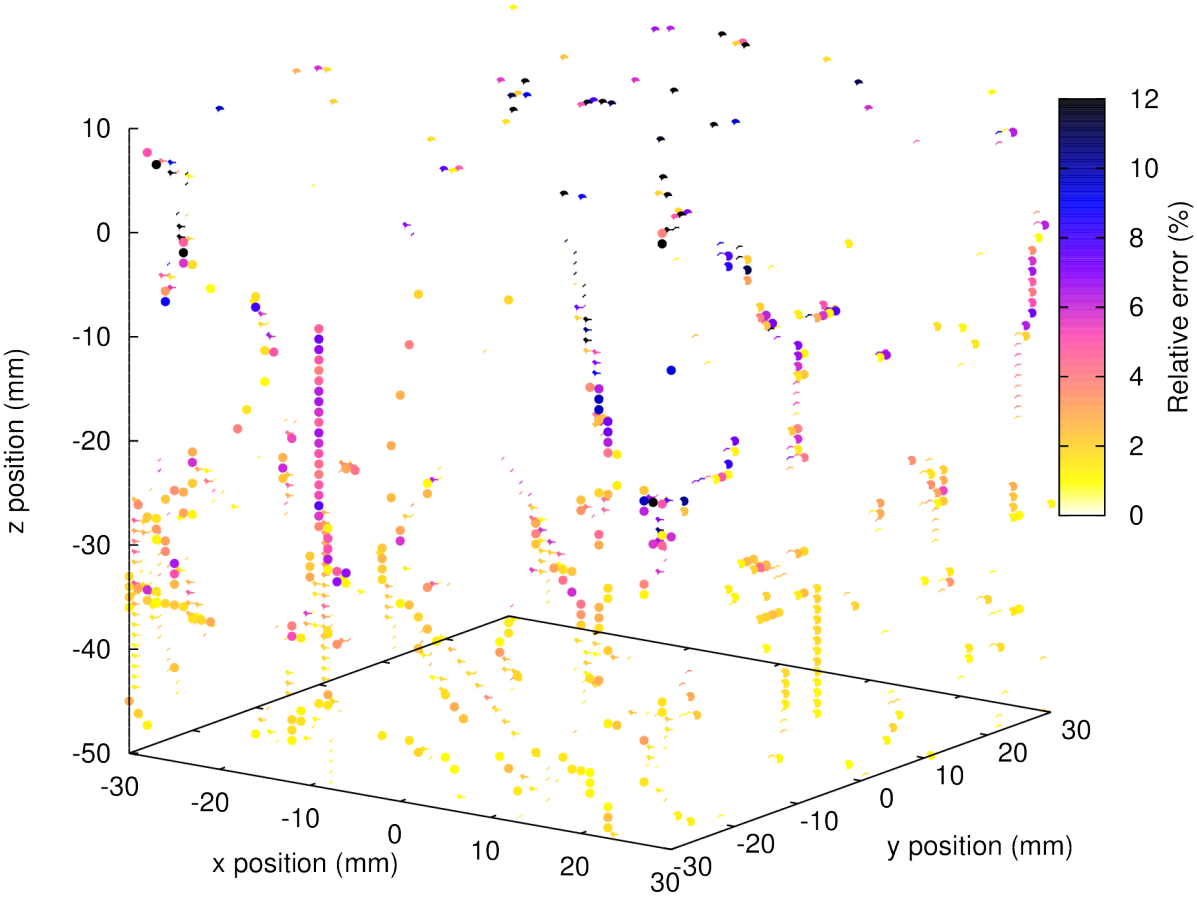

Figure 4.2 Voxel-per-voxel relative difference between single- and double-precision tracking computations, in a subset of the simulated volume (test case 2 with 58 sources). Darker spots mark larger relative differences. The sources are located at the bottom of this volume.

difference between each contribution and the total energy deposited is effectively eliminated, at the cost of a small increase in computation time.

For the other two components of the program – particle tracking and physics – the difference pattern is very distinct from the energy accumulation. Figure 4.2 shows that for the 58-source test case, some differences do occur, but mainly from single particle trajectories. These relative differences are much more visible when the rays are farther from the sources, which are located at lower values along the z axis. This is because the dose is lower in voxels farther from the sources, makings even small dose differences more noticeable. This pattern is less noticeable in the single-source test case, because of the lower number of particles going through a source seed, which requires much more complex computations than inside a uniform material. The comparison of the physics component yields a pattern very similar to that of the tracking component, but with less intensity.

Table 4.1 summarizes the differences between single and double precision for the tracking and physics components of `bGPUMCD` by displaying the proportion of particles which terminate their trajectory in the wrong voxel and of those which undergo the wrong interactions. These two metrics suggest that the two components are closely related, since photon positioning affects the interactions occurring and vice versa. It also shows that the tracking component plays a more important role in terms of precision of the simulation.

Tableau 4.1 Proportion of particles which finish in the wrong voxel and proportion which are subject to the wrong interaction when executing the tracking and physics components of `bGPUMCD` in single precision rather than double.

	Single source		Multiple sources	
	Wrong end voxel (%)	Wrong interaction (%)	Wrong end voxel (%)	Wrong interaction (%)
Tracking	0.015	0.0045	0.035	0.0121
Physics	0.005	0.0012	0.007	0.0026

To provide a better insight about the significance of the errors observed above, figure 4.3 and 4.4 show histograms of error size for each program component, for both test cases.

As was clearly shown in figure 4.1, the energy accumulation component for the single-source case contains very large errors in a small proportion of voxels. For the multiple-source case, these errors are generally smaller and occur in less voxels. In the rest of the program, errors for the single-source case are very small and occur only occasionally, in a negligible number of voxels. For the multiple source case, errors of up to 0.1% are observed in 55% of voxels for the tracking component et 20% for the physics component.

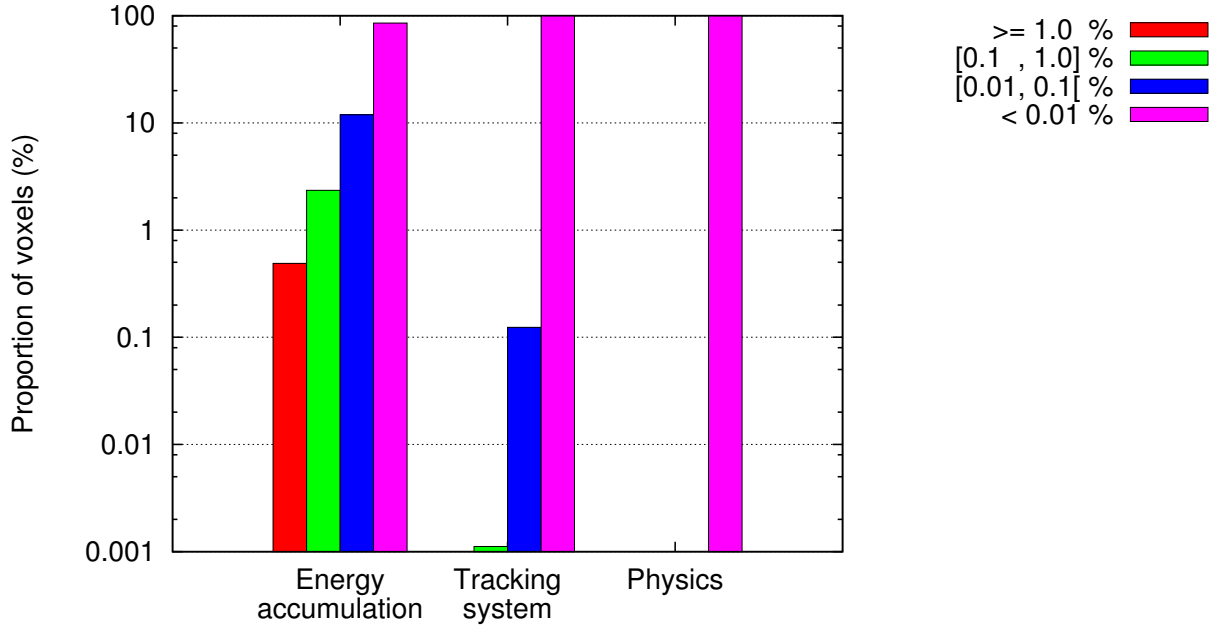

Figure 4.3 Relative error size distribution between single- and double-precision computations for the single-source test scenario, by program component. This is an average of 8 comparisons.

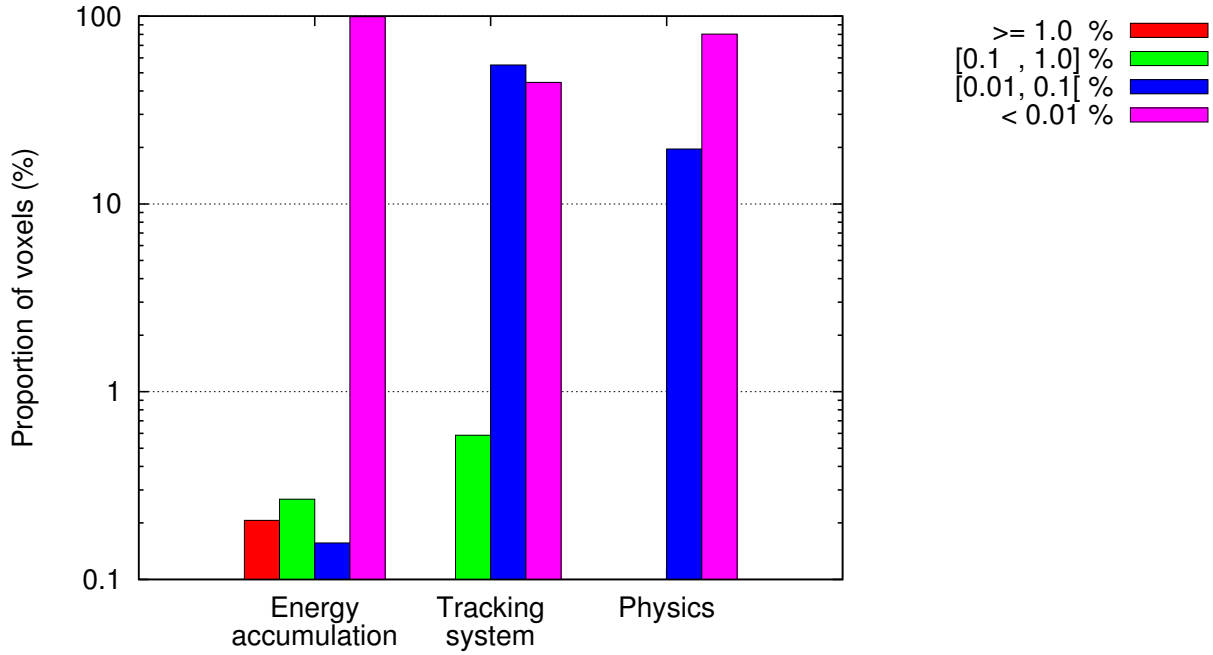

Figure 4.4 Dose difference in % between single- and double-precision computations for the multiple-source test scenario, by program component. This is an average of 8 comparisons.

4.4.2 Fast math option

Figure 4.5 shows the differences that results from using the fast math option in each test case. The figure shows very little difference between a simulation run with the intrinsic

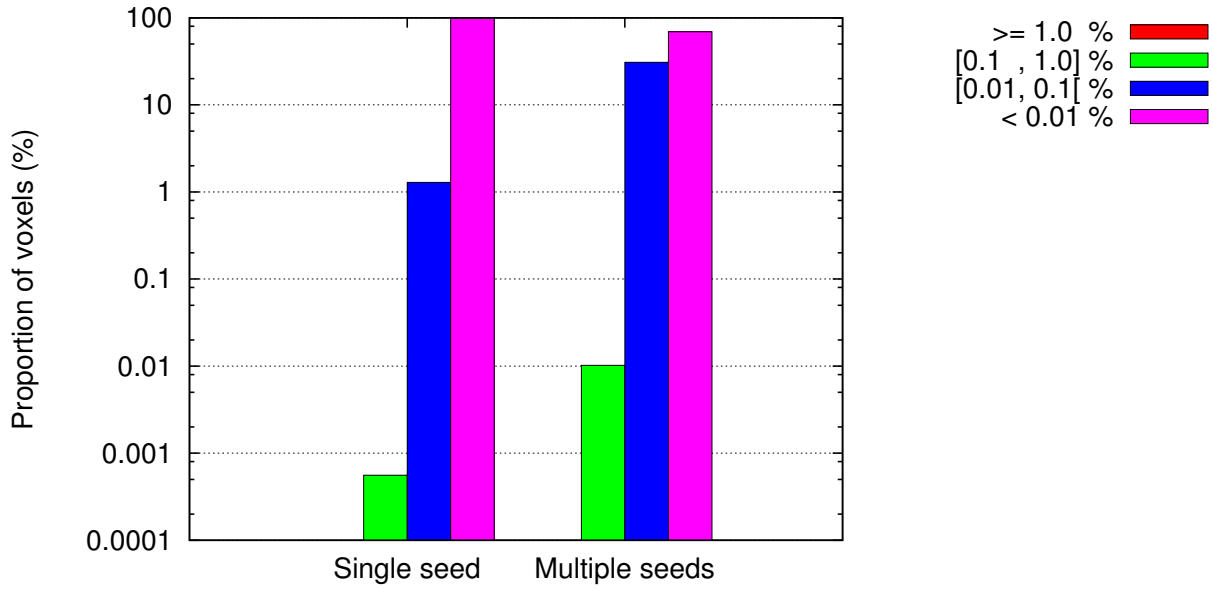

Figure 4.5 Difference between executions with regular math library functions and with CUDA intrinsic functions. This is an average of 8 comparisons.

functions from one with the library functions. This suggests that intrinsic functions are safe to use for this particular application. However, performing the test revealed an underlying fault in `bGPUMCD`. By using more relaxed requirements for trigonometric functions, the computations allow for a result of negative zero (the floating point representation of zero can either be positive or negative). This number can be used in additions, multiplications and comparisons just like the default positive zero, but any number divided by it will become a special floating point value representing infinity with an opposite sign, unlike a positive zero which does not change the sign. Some computations in the program relied on the fact that a positive number divided by zero becomes infinity and used that result in a comparison. This in turn allowed some voxels to end up with an infinite energy and caused the final

dose distribution to contain a small, scattered set of invalid data. To resolve this issue, it is sufficient to simply check for a value of zero and, instead of performing the division, set the result to the largest possible floating point value.

4.4.3 Tracking system

Problems related to precision arise in some cases, when a particle is near an intersection point with a quadric surface, but on a trajectory almost tangential to it. In such cases, multiple cancellation-born errors occur and lead to inaccurate results. In line 2 of Alg. 4.1, the computed dot product between the trajectory and the surface normal is close to zero, implying a cancellation during the addition. In line 3, the value computed approaches zero as the particle approaches the surface. Variables B and C are then used in line 4 to compute another result very close to zero. Finally, this leads to another cancellation in line 8, where both subtracted numbers in the numerator are close to zero.

All these cancellation-related errors become apparent when the result of the function has an order of magnitude of 10^{-6} units or less because it is added to the position, which takes values superior to 1 unit in most of the volume. Such a large difference between the two operands of the addition creates two problems.

The first one occurs because the distance, which is small and relatively inaccurate, is added along a specific direction. It is thus separated into three components (x, y and z) by multiplying it with the direction vector. Each of these components contain a different error, resulting from the error already present in the distance and the multiplication itself. When they are added to the three components of the position, the particle is thus displaced slightly off-course.

The second problem can have a more profound effect. If one (or more) of the three components is sufficiently small compared to the position to which it is added, it can be completely ignored, in a way similar to the energy accumulation issue (4.4.1). As a consequence, the new position locates the particle on the same side of the surface as it already was. In the context of the track length computation, which finds iteratively each surface crossed by the particle, this means that on the following iteration it will find the same surface. Since the new distance to that surface will also be very small, the same problem might occur and the particle might stay on the same side again.

This phenomenon may go on for many iterations – up to hundreds of thousands in some extreme cases. Each time, the particle drifts slightly from its trajectory, gliding along the surface it should be crossing instead. The iterations end either when the particle goes completely around the object or when the computations place it on the other side of the surface. In both cases, the particle ends up aside from its correct position and the distance to its next

interaction point is skewed. Another consequence is that, since threads in a warp execute in lock-step, the effect of one such problematic particle on execution time is amplified : even one particle in a hundred thousand can cause a tenfold increase in execution time.

Another problem, that occurs in some rare cases, is that `computeIntersection` can yield results that are not inaccurate but simply wrong. The function can return an intersection while the particle passes beside the surface (false positive) or an absence of intersection while the particle intersects the surface (false negative). These errors occur when the cancellation-related errors in Alg. 4.1 at lines 2 and 3 result in a radical with the wrong sign at line 4. As with the issues described above, this can cause the tracking system to output a skewed distance, since it will assume the particle passes through the wrong materials.

The cancellation-related errors cannot be completely avoided due to the nature of the computation : it is necessary to determine on which side of the surface the particle falls. However, their effects can be greatly reduced by verifying whether adding the distance found to the particle's position actually places the particle on or immediately beyond the surface and by adjusting the distance if needed.

The proposed solution is to implement an iterative algorithm which uses Newton's gradient descent method. This algorithm effectively corrects most problems. First, by ensuring that the particle reaches the surface (when there actually is an intersection), it guarantees that it will not need to compute the same particle/surface intersection again. Second, by incrementally correcting the distance until the particle changes sides, it finds a more accurate distance without the need to increase the precision of computations.

Figure 4.6 presents a profile of error sizes in the results from the two versions of `computeIntersection` and from the original version when computed with double-precision operations. It shows that the new implementation practically eliminates false positives and false negatives. Another advantage is that it also eliminates absolute errors above 10^{-6}, which is an approximated limit on the distance increment that can be computed when using a unit direction vector and the size of the error that leads to the issues discussed above – a repetition of the distance computation from a particle to a certain quadric surface and a slowdown in execution time. The rest of this histogram is similar to that of the original algorithm.

The third histogram of figure 4.6 presents the error rate of the original function executed in double precision. The double-precision version takes approximately 7 times longer to execute and also suffers, to a lesser extent, from the slowdown discussed above caused by inaccuracies. The interesting point to note is that the higher precision does not reduce the number of false positives and negatives at all.

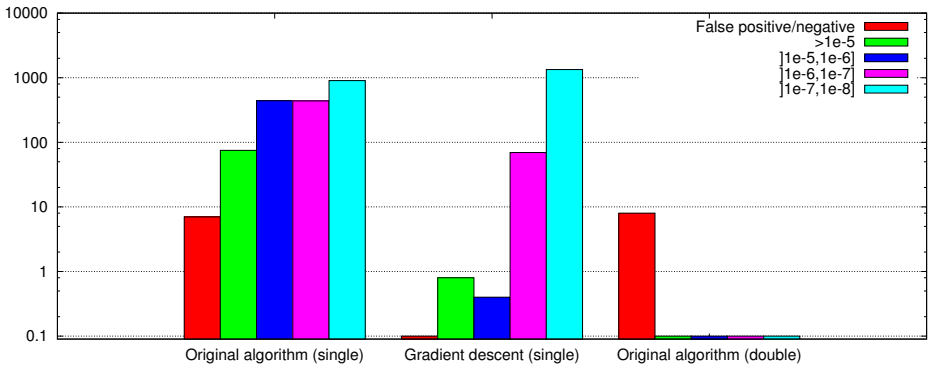

Figure 4.6 Number of calls to `computeIntersection` per million that result in a certain error, for each version of the function.

4.4.4 Error injection

For the purposes of the tests discussed below, results were considered identical when no voxel contained a relative difference greater than 0.05% when compared to the distribution without any introduced error. The results were compared by computing the distribution of error sizes in the same way as for the histograms presented in sections 4.4.1 and 4.4.2.

It was determined that a modification of the last 12 bits of most physical data (densities, cross sections, form factors, probability densities), corresponding to a relative error of about 2.4×10^{-4}, was necessary to affect the final result of a simulation. Since the physical data are read as an input to the program and their representational error (around 1.2×10^{-7}) is lower than this value, they are definitely not a source of numerical error in `bGPUMCD`.

For remaining error injection tests, the error size that would visibly affect the final results varied considerably from about 1×10^{-6} units to about 3×10^{-4} units, depending on the function result or parameter being modified. The most sensitive function was `computeIntersection`. However, it was shown earlier that the results from the new algorithm have errors above 1×10^{-6} less than once in a million calls (see figure 4.6). It is thus safe to assume that this function does not introduce any notable numerical error in the final results.

The other relevant geometric functions tested this way are used to compute the length of a particles trajectory through each voxel and to rotate its direction vector following an interaction. Both their outputs contain a numerical error far inferior to the error that would be necessary for them to affect the results of a simulation.

The remaining components of the program are particle interactions. They were tested indirectly, through the modification of individual particle properties – position, direction and energy – and of physical data. Interactions have a more complex effect on particles than

geometric functions, since they modify both their energy and direction, which in turn have an impact on a particle's future position and interactions. This makes it more difficult to ascertain whether they are free of numerical errors. However, their computations are much less complex since they are mostly based on lookup tables (built from physical data) and thus very resilient to errors : they would have to be large enough to cause a change in a table slot, otherwise they would have no effect at all. Numerical errors in particle interactions would rather have to come from the inputs that affect these interactions. These inputs are the physical data, which have already been established as numerical error-free, photon position and random numbers. The position is itself determined by geometric functions, which do not generate errors that are large enough to affect simulation results. The random numbers are generated from integer numbers, which means they do not contain any floating point-related error. It thus seems reasonable to assume that, under these considerations, the particle interaction component of `bGPUMCD` does not introduce any numerical error large enough to be detected in the final results.

4.5 Conclusion

The results summarized in this paper are useful because many of the errors found in `bGPUMCD` computations are not caused by poor algorithm design or programming, but stem from the way simple operations are implemented by the hardware. The tests run on `bGPUMCD` have revealed two errors related to the floating point representation of real numbers and suggest that no other major numerical error remains in the program.

The first error, a lower dose in voxels close to the radiation source, was detected because of its clear effect on the final result and because it was corrected by performing the computations in double precision. Its cause was not a progressive loss of precision by rounding, as expected, but a complete loss of some energy deposits that were too small to be added to the total.

The second error, inaccurate distances between particles and surfaces, had no significant effect on the result of the simulation. It was only detected because of its side effects on execution time. Its cause was a series of cancellation-related errors which could have consequences on the execution flow, either by returning a distance that was too small for a particle to reach a surface, or by erroneously hitting or missing the surface.

The error injection tests also established the minimum size of errors in various parts of the program that would be detected in the final results. These bounds were useful to determine that, at least in the tested parts, no numerical error was sufficiently large to have an effect on the simulation. Furthermore, they showed that rounding errors during energy accumulation in every voxel were not large enough to affect the result, except in a few extreme cases. These

extreme cases are avoided by using multiple buffers to accumulate energy.

As shown above, relatively simple operations like a summation and a linear algebra equation may introduce errors when using a floating point representation. These types of computations arise in many programs other than `bGPUMCD`, which may or may not be Monte Carlo simulations. When precision is important, careful consideration must therefore be given to such possibilities.

Acknowledgment

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by Elekta Ltd - Canada. Some calculations were performed on the high-performance computing infrastructures of Compute Canada and Calcul Québec.

CHAPITRE 5

DÉMARCHES ADDITIONNELLES

Ce chapitre décrit les démarches effectuées pour atteindre les objectifs (section 2.4) qui n'ont pas été mentionnées dans l'article du chapitre précédent, soit par souci de synthèse, soit parce que la démarche est suffisamment différente du contenu de l'article pour mériter une section séparée.

Les tests non mentionnés dans l'article consistent à vérifier que la cause de l'erreur dans l'accumulation de l'énergie est bien causée par la perte de certaines contributions individuelles et à examiner l'effet de différentes résolutions de voxel sur les erreurs numériques. Quelques détails sont fournis sur les tests déjà effectués, plus spécifiquement sur leur implantation, de même que sur les résultats obtenus, en particulier pour les calculs en précision double et pour l'injection d'erreur. Enfin, une méthode de détection automatique est implantée et comparée aux résultats obtenus par les tests précédents.

5.1 Détails sur l'article

5.1.1 Passage entre précision simple et double

Pour faciliter l'exécution des tests et, surtout, leur reproductibilité, trois nouveaux types de données ont été introduits : `result_t` pour les variables impliquées dans l'accumulation de l'énergie (convertie en dose pour obtenir le résultat de la simulation), `trace_t` pour les variables utilisées dans le tracé de rayon et `real_t` pour les autres variables à virgule flottante. Une définition de type associe chacun de ces types avec le type `float` ou `double` lors de la compilation à l'aide de la directive `typedef`. Les déclarations de variables et de constantes `float` ont toutes été remplacées par un des trois types (le type `double` n'était pas utilisé par `bGPUMCD`), selon la division décrite à la section 3.3.1, à quelques exceptions près. Les données stockées sous forme de texture sont demeurées en précision simple étant donné que seuls les types d'une taille de 4 octets sont acceptés par les fonctions de texture de CUDA. Une solution utilisant des types composés (déjà définis par CUDA) serait possible, mais nécessiterait des changements considérables pour chaque appel à des fonctions de texture. De plus, comme les divers tests effectués le démontreront, la précision simple ou double des données impliquées n'a pas d'impact sur le résultat final du programme.

Deux modifications supplémentaires étaient nécessaires pour que `bGPUMCD` fonctionne en double précision. La librairie mathématique de CUDA contient deux ensembles de fonctions

pour nombres à virgule flottante, en précision simple et double. Les appels à ces fonctions ont été modifiés pour que la version simple ou double soit exécutée, selon le cas. Par ailleurs, CUDA définit des types vecteurs de deux, trois ou quatre composants du même type. Les fonctions utilitaires pour définir les opérateurs et effectuer des opérations de base sur ces vecteurs qui sont fournies par la trousse de développement sont cependant limitées aux entiers et aux nombres à virgule flottante de simple précision. Pour chacune de ces fonctions, une version en double précision a donc été implantée. Enfin, la librairie CUDA ne fournit aucune fonction atomique en double précision, en raison de l'impact important de celles-ci sur les temps de calcul. La fonction proposée par le guide de programmation (NVIDIA, 2012) a donc été implantée pour l'addition (la seule fonction atomique utilisée par `bGPUMCD`).

Un test supplémentaire, omis de l'article, a été effectué à la suite de la correction de l'erreur dans l'accumulation de l'énergie. L'impact de la séparation en deux étapes de la sommation de la dose dans chaque voxel a été étudié en exécutant le programme avec différents nombres de particules par noyau.

5.1.2 Injection d'erreur

Cette section décrit plus en détail les tests d'injection d'erreur mentionnés au chapitre 4.

L'ajout d'erreurs plus grandes que celles déjà existantes permet d'extrapoler leur impact ainsi que de déterminer les sections du programme qui y sont plus sensibles. Deux types d'erreurs sont introduites : des erreurs d'amplitude aléatoire, qui simulent des erreurs numériques qui pourraient être générées lors d'une simulation, et des erreurs d'amplitude fixe, qui servent de témoin pour vérifier que le changement a bel et bien un impact sur le programme. L'impact sur le résultat de la simulation est déterminé à l'aide de la différence de dose (voir la section 3.2.1).

L'amplitude des erreurs est déterminée à l'aide d'un générateur de nombres pseudo-aléatoires identique à celui de `bGPUMCD`, mais implémenté et initialisé complètement séparément, afin d'éviter les interactions autres que celles causées par les erreurs. La seule différence est reliée à son utilisation : l'état du générateur n'est pas sauvegardé localement par les fils d'exécution. Lors de chaque appel au générateur, l'identifiant unique de chaque fil est calculé, puis utilisé pour récupérer l'état du générateur pour ce fil dans la mémoire globale et déterminer le prochain nombre. Cette solution permet de faire un simple appel à une fonction sans paramètre pour obtenir un nombre aléatoire, mais est cependant plus lente que la méthode adoptée par `bGPUMCD`.

Tous les tests d'injection d'erreur ont été effectués sur le deuxième cas décrit à la section 4.3.1, c'est-à-dire un cube d'eau de 30 cm de côté, des voxels cubiques de 0.1 cm de côté et 58 sources de radiation.

Frontières des voxels

Pour évaluer l'effet d'associer les particules avec un voxel autre que celui où elles se trouvent, une erreur est introduite dans la fonction qui détermine le voxel où se situe une particule, afin que lorsqu'elle se trouve proche d'une limite entre deux voxels, elle soit assignée au voxel voisin. L'erreur, proportionnelle à la dimension des voxels, est appliquée sur chaque composante de la position des particules, au moment de déterminer dans quel voxel elles se situent. Cette modification simule l'accumulation de l'erreur sur la position des particules au cours de l'ensemble des calculs.

Position des particules

Pour étudier de façon plus générale l'importance de la position exacte des particules, une erreur est ajoutée sur chaque composante de la position avant chaque interaction. Cette modification simule l'erreur accumulée depuis le début de l'exécution par la création des particules et par tous les calculs reliés à des changements de position. Contrairement à l'erreur décrite précédemment, celle-ci est conservée pour le reste de la simulation.

Direction des particules

En ce qui concerne la direction des particules, deux types d'erreurs peuvent être examinés. Le premier survient lors de l'initialisation des particules. Leur direction est déterminée à partir de deux angles aléatoires en coordonnées sphériques, qui sont convertis en coordonnées cartésiennes à l'aide de plusieurs appels à des fonctions trigonométriques. L'erreur introduite à ce moment simule l'ensemble des erreurs causées par la génération des angles et par leur conversion. Le second type d'erreur provient des interactions. La modification de la direction après chaque interaction simule l'erreur accumulée depuis le début de la simulation, à la fois à cause de la création des particules et à cause des interactions.

Dans les deux cas, l'erreur ajoutée est calculée à partir de deux angles, qui sont utilisés pour appliquer une rotation sur le vecteur de direction des particules. La précision de la fonction utilisée pour calculer la rotation (`rotateDirectionVector`) est évaluée séparément (voir section 5.1.3).

Interactions

Étant donné que les particules secondaires ne sont pas simulées, les seuls effets d'une interaction sur un photon sont une perte d'énergie et un changement de direction. L'erreur générale sur la direction est examinée de la manière décrite ci-dessus, de la même façon pour

toutes les interactions puisqu'il s'agit à chaque fois d'une rotation de vecteur. Par contre, la variation d'énergie est traitée différemment pour chaque interaction.

Pour l'effet Compton, l'énergie diffusée correspond à une fraction de l'énergie initiale du photon et est calculée à partir de l'énergie initiale, de constantes physiques et de nombres aléatoires. L'erreur est appliquée sur cette fraction, afin que son impact soit cohérent avec le reste des calculs. Les calculs reliés à cette interaction sont aussi affectés par les modifications aux constantes physiques, décrites dans les sous-sections suivantes.

Pour l'effet photoélectrique, l'énergie varie par certaines quantités fixes, qui sont déterminées par le matériau dans lequel le photon se trouve et par une probabilité. Les modifications introduites sont les mêmes que pour les autres données physiques et sont appliquées sur les niveaux d'énergie, les sections efficaces et les valeurs de probabilité (qui sont écrites directement dans le code) pour simuler l'erreur de représentation des données. La déviation du photon est calculée complètement indépendamment du changement d'énergie et est testée séparément.

L'unique effet de la diffusion de Rayleigh dans `bGPUMCD` est un changement dans la direction du photon, qui dépend en partie du facteur de diffusion du photon. Ainsi, aucune erreur autre que celle ajoutée à la direction et aux facteurs de diffusion n'est examinée pour cette interaction.

Énergie des particules

L'énergie des particules varie à travers plusieurs mécanismes, implantés dans les fonctions d'interactions. Les erreurs ajoutées qui modifient l'énergie sont donc intégrées dans les tests sur les interactions.

Données physiques

Les données physiques, incluant les valeurs de densité, de section efficace et de facteur de diffusion, sont lues lors de l'initialisation et utilisées constamment pour calculer les probabilités d'interaction et pour déterminer leur effet. Les mêmes données sont utilisées par chacun des fils d'exécution. Ainsi, la modification est introduite une seule fois, après la lecture des valeurs et leur interpolation s'il y a lieu, pour simuler l'erreur de représentation de ces valeurs. L'amplitude de l'erreur est calculée en fonction de la valeur à altérer, pour qu'elle corresponde à un certain nombre de bits modifiés ; il s'agit donc d'une erreur relative, contrairement aux autres erreurs introduites.

Distance parcourue

Pour simuler les particules, `bGPUMCD` utilise une méthode itérative, en calculant à chaque étape la distance entre la position de la particule et celle de sa prochaine interaction (le libre parcours), puis en calculant les effets de cette interaction. Le calcul de la distance utilise l'algorithme de Woodcock, décrit dans Hissoiny *et al.* (2011a), lorsque la particule ne traverse aucune source. Cependant, lorsque la particule traverse une source, la distance de base, calculée selon l'algorithme de Woodcock, est modulée par la densité des différents matériaux qui composent la source, ce qui nécessite un calcul exact de la distance avec chacun d'eux. Le calcul de la distance entre la particule et chaque transition entre deux matériaux de la source est effectué par la fonction `calculeIntersection`, décrite à la section 4.4.3. Une modification est introduite dans le résultat de cette fonction pour simuler les erreurs accumulées par l'implantation de l'équation utilisée. Lors d'un autre test, une modification est introduite dans la distance entre deux interactions, pour simuler l'erreur résultant de l'utilisation de `calculeIntersection` en conjonction avec l'algorithme de Woodcock. Finalement, une fois que la distance entre deux interactions est déterminée, il faut calculer la distance parcourue dans chacun des voxels sur la trajectoire du photon. L'énergie déposée par la particule dans chaque voxel est proportionnelle à cette distance, selon l'estimateur de parcours linéaire (Williamson, 1987). Une modification y est donc introduite pour simuler l'erreur générée par son calcul.

Accumulation de l'énergie

L'objectif de cette dernière modification est de vérifier que la grande différence observée entre les précisions simple et double lors de l'accumulation de l'énergie dans chaque voxel est bien causée par le rejet de valeurs trop petites par rapport à la somme totale. La modification est appliquée uniquement aux calculs en double précision et consiste à ignorer toute valeur à ajouter à un voxel si elle est 2^{25} fois inférieure à l'énergie totale contenue dans ce voxel. Lorsque cette valeur se trouve entre 2^{25} et 2^{24} fois inférieure à l'énergie du voxel, elle est arrondie à $1/2^{24}$ fois l'énergie, pour simuler l'arrondi obtenu lors de l'opération en simple précision. Le reste de ce texte fait référence à cette modification en tant qu'accumulation en double précision tronquée.

5.1.3 Fonctions géométriques

Seul le test de la fonction `calculeIntersection` a été présenté au chapitre 4. Les autres fonctions testées sont `changeDirection`, qui effectue la rotation du vecteur direction des particules après chaque interaction et `calculeSegment`, qui calcule la distance parcourue par

une particule dans chacun des voxels le long de sa trajectoire. Ces trois fonctions sont les seules qui requièrent plus que quelques opérations simples sans avoir recours à des nombres aléatoires.

`changeDirection` modifie le vecteur direction d'une particule en utilisant un système de référence dans lequel ce vecteur a une valeur de 1 en direction Z et de 0 dans les deux autres directions, en appliquant la rotation dans ce système de référence, puis en transposant le nouveau vecteur dans le système de référence initial. Ces calculs nécessitent des racines carrées et des divisions, mais aucun appel à des fonctions trigonométriques étant donné qu'ils utilisent des valeurs précalculées des sinus et cosinus des angles de rotation.

`bGPUMCD` utilise l'algorithme de Siddon (1985) pour calculer la longueur du trajet des particules dans les voxels. Cet algorithme considère en fait le trajet des particules entre des plans parallèles plutôt que des voxels pour simplifier les calculs. La fonction `calculeSegment` a pour rôle de calculer la distance entre deux plans qui intersectent les points d'entrée et de sortie d'une particule dans un voxel. Les calculs nécessitent plusieurs conversions entre les coordonnées cartésiennes et les coordonnées en voxels, à l'aide d'additions, de multiplications et de divisions.

Pour chacune de ces fonctions, une version a été implantée en Python, en utilisant la librairie d'arithmétique flottante de précision arbitraire *mpmath* (Johansson *et al.*, 2013). Cette librairie permet d'effectuer les calculs avec une grande précision (100 décimales lors de ces tests), dans le but de les comparer avec les résultats obtenus par la fonction sur GPU.

Pour obtenir des cas tests réalistes, des appels aux fonctions testées ont été sélectionnés de façon aléatoire et tous leurs paramètres ainsi que leurs résultats ont été enregistrés dans un tampon, puis transférés sur l'hôte dans un fichier pour être lus par les programmes de test. Pour s'assurer que les quantités exactes soient utilisées, les nombres à virgule flottante ont été écrits en format hexadécimal.

5.2 Discrétisation en voxels

L'article présenté au chapitre 4 répond aux objectifs 1 et 2 (voir section 2.4.1) en décrivant les résultats obtenus lorsque les calculs sont effectués en double précision ou avec les fonctions intrinsèques. Les effets de la discrétisation en voxels (objectif 3) sur les erreurs causées par les nombres à virgule flottante sont plus difficiles à déterminer en raison des comparaisons nécessaires.

La comparaison entre deux ensembles de résultats est plus complexe lorsque les distributions n'ont pas exactement la même résolution (en nombre de voxels par unité de longueur, qui peut varier selon les axes). En effet, il n'est pas possible de comparer les valeurs voxel

par voxel dans un tel cas. Deux solutions ont été envisagées pour résoudre ce problème. La première consiste à redistribuer la dose d'un des deux ensembles de résultats afin qu'il ait une résolution identique à l'autre. Ce changement permet d'effectuer les comparaisons décrites précédemment (voir section 3.2), comme si les simulations avaient été réalisées avec la même discrétisation. La seconde solution, celle adoptée lors de ce travail, consiste à comparer chaque voxel de la distribution référence avec tous les voxels de la distribution évaluée dont une partie du volume se superpose au voxel référence et à pondérer chaque différence par la proportion du volume référence qui est occupée par ces voxels. Cette méthode est illustrée en deux dimensions dans la figure 5.1.

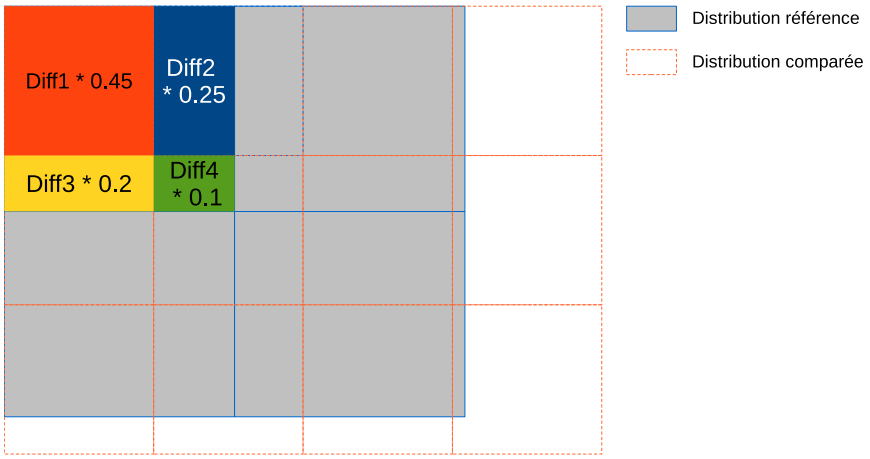

Figure 5.1 Schéma 2D de la comparaison de deux distributions avec différentes résolutions. La différence calculée pour le voxel supérieur gauche de la distribution référence correspond à la somme des différences avec chaque voxel de la distribution comparée qui le chevauchent, pondérées par la surface que chacun de ces voxels couvre sur le voxel référence.

Deux situations différentes seront étudiées en ce qui concerne la résolution de l'espace simulé. Dans le premier cas, les voxels ont la même résolution le long des trois axes. Le deuxième cas testé utilise des voxels non cubiques, afin d'examiner les différences possibles entre les axes.

5.3 Infrastructure de test

Un ensemble de scripts ont été écrits pour faciliter l'exécution des tests décrits précédemment et pour aider à l'organisation des résultats obtenus. Ces scripts gèrent la compilation et l'exécution du programme avec différents paramètres et placent les fichiers de résultats dans des dossiers séparés selon le test effectué. Les différents paramètres servant à choisir la précision de différentes variables, les options particulières du compilateur `nvcc` et les différentes

erreurs à introduire ainsi que leurs caractéristiques (amplitude, signe, caractère aléatoire ou non) ont été directement intégrés dans le Makefile et dans le code, à l'aide de directive de compilation conditionnelle. Les scripts de test gèrent également les appels au programme permettant d'effectuer les comparaisons entre différentes distributions de dose pour produire les résultats présentés au prochain chapitre. En automatisant l'exécution des tests et des comparaisons, ces scripts assurent la cohérence des différents tests et résultats ainsi que leur reproductibilité.

5.4 Arithmétique stochastique discrète

Pour tenter de détecter de façon automatique les erreurs causées par la représentation en virgule flottante, une méthode utilisant l'arithmétique stochastique discrète (DSA) semblable à celle de la bibliothèque CADNA (Jézéquel et Chesneaux, 2008) a été implantée. Un module décrivant un nouveau type composé pour remplacer les nombres à virgule flottante a été écrit. Ce type contient N exemplaires de la variable en virgule flottante qu'il représente. Les opérations sur cette variable sont appliquées avec un mode d'arrondi différent pour chaque exemplaire, choisi aléatoirement entre vers l'infini positif ou négatif. Seuls ces deux modes d'arrondi sont utilisés pour garantir que les différents échantillons soient supérieurs et inférieurs à la valeur exacte de la variable. Chaque exécution du programme donne donc lieu à une série de N scénarios possibles pour toutes les valeurs en virgule flottante utilisées. En vérifiant au cours de l'exécution certaines propriétés des variables composées, il est théoriquement possible de trouver des opérations problématiques dans le code source. Par exemple, si une instruction conditionnelle effectue une comparaison avec une variable, mais que certains exemplaires de cette variable sont positifs et d'autres négatifs, il n'est pas possible de choisir quelle condition est vraie. Le fonctionnement du module DSA sera décrit plus en détail au cours des sections suivantes, en ce qui a trait aux modifications nécessaires, aux événements détectés et à la méthode utilisée pour rapporter ces événements.

5.4.1 Définition des opérations

Pour limiter les changements au code source de `bGPUMCD` et ainsi créer un module le plus général possible, les opérateurs mathématiques ont été définis pour le nouveau type `dsa_t`. Pour chaque opérateur, l'opération est exécutée N fois, à chaque fois avec un mode d'arrondi sélectionné aléatoirement, à l'aide d'un générateur de nombres aléatoires semblable à celui utilisé pour l'injection d'erreurs (voir section 5.1.2). Les autres fonctions mathématiques utilisées, telles que `sin` et `cos`, ainsi que les opérations sur des vecteurs ont été implantées pour le type `dsa_t` de la même façon.

L'opération `atomicAdd`, la seule opération atomique utilisée par `bGPUMCD`, a été implantée différemment. Chaque exemplaire de la variable est traité séparément, ce qui est en accord avec l'idée que chaque exécution peut donner lieu à un ordre différent des opérations. Cependant, le mode d'arrondi utilisé est toujours vers le nombre le plus près, afin d'obtenir des résultats similaires à ceux d'une exécution normale de `bGPUMCD`. L'utilisation d'autres modes d'arrondi entraîne des différences trop importantes dans l'accumulation de l'énergie pour que les résultats du module DSA soient réalistes dans ces conditions. Comme il a été démontré à la section 4.4.1, les erreurs qui ont lieu lors de cette sommation s'annulent en grande partie.

Ainsi, il suffit de remplacer chaque déclaration de nombre à virgule flottante par le type `dsa_t` pour pouvoir exécuter le programme en utilisant le module DSA. Ce remplacement est particulièrement simple dans le cas présent étant donné que tous les nombres à virgule flottante sont déclarés avec un des types décrits en section 5.1.1

5.4.2 Événements détectés

Le mécanisme principal par lequel les erreurs peuvent être détectées consiste à calculer le nombre de bits valides d'un nombre. En sélectionnant de façon aléatoire le mode d'arrondi lors d'un calcul, l'erreur possible est amplifiée. Si la valeur de l'erreur est considérée comme une variable aléatoire de distribution gaussienne – ce qui est le cas, à toutes fins utiles (Li *et al.*, 2011) – il suffit d'utiliser les échantillons fournis par les différentes valeurs contenues dans une variable de type `dsa_t` pour l'estimer. Cette méthode permet de connaître le nombre de bits exacts dans un nombre à virgule flottante.

Lorsqu'un nombre ne contient aucun bit exact, il est considéré comme un "zéro informatique" (*computational zero*) ; il n'est pas possible de déterminer sa valeur, ni même son signe. Si les N éléments d'une variable sont exactement 0, cette variable est également considérée comme un zéro informatique. Ce concept permet d'identifier plusieurs situations problématiques.

Expressions conditionnelles

Toutes les expressions conditionnelles peuvent être converties sous une forme équivalente à une comparaison avec zéro. Si la variable comparée est un zéro informatique, il n'est pas possible de déterminer le résultat de l'expression. Dans un tel cas, pour que l'exécution du programme continue, la branche choisie est celle qui correspond à la moyenne des N éléments.

Division par zéro

En arithmétique flottante, le résultat d'une division par zéro est l'infinie, avec le même signe que le nombre divisé. Avec le module DSA, une division par un zéro informatique est signalée comme une erreur.

Annulation

La soustraction de deux nombres très proches résulte en une perte de précision, tel que décrit à la section 2.1. Une erreur est signalée lorsque la différence entre le nombre de bits exacts du résultat d'une soustraction et celui du moins précis des opérandes dépasse un certain seuil.

Autres erreurs

D'autres situations peuvent survenir dans lesquelles un nombre trop imprécis pose problème. Le module DSA peut signaler des erreurs lorsque les deux facteurs d'une multiplication sont zéro, lorsque la base ou l'exposant dans la fonction `pow` est égal à zéro et lorsque les diverses fonctions mathématiques (`sin`, `cos`, etc.) ont zéro comme opérande.

5.4.3 Enregistrement des événements

C'est au niveau de l'enregistrement des événements détectés que le module implanté ici se distingue de la bibliothèque CADNA. Cette dernière utilise un compteur pour chaque type d'erreur et indique le nombre total de chaque événement présent dans le programme. L'utilisateur doit alors utiliser une méthode externe, par exemple un débogueur, pour déterminer exactement les lignes de code qui ont causé ces erreurs en obtenant la trace du programme au moment de l'erreur.

Cette méthode est difficile à appliquer à un programme sur GPU pour plusieurs raisons. Tout d'abord, le grand nombre de fils exécutant exactement le même code signifie qu'une erreur dans une ligne de code source peut être signalée des millions de fois. Inversement, une erreur peut se produire sous des conditions très rares et affecter un seul fil d'exécution, sans avoir d'effet visible sur le résultat global du programme. Il est donc nécessaire de connaître l'importance relative des erreurs détectées. Enfin, l'utilisation d'un débogueur peut être compliquée lorsque plusieurs erreurs différentes sont présentes dans le programme. En effet, un point d'arrêt conditionnel serait atteint chaque fois qu'une erreur se produit (ce qui peut arriver dans des milliers de fils simultanément) et l'utilisateur doit déterminer chaque fois s'il s'agit d'une nouvelle erreur ou si elle a déjà été examinée.

Le fonctionnement général du module DSA consiste à ajouter une vérification lors de chaque opération implantée, pour déterminer si une erreur y est présente. Lorsqu'il y a une erreur, une fonction (`enregistrerErreur`) est appelée pour incrémenter un compteur pour ce type d'erreur, associé au nom de fichier et au numéro de la ligne de code. Le résultat d'une exécution donne ainsi une liste des catégories d'erreurs détectées et, pour chaque catégorie, les noms de fichiers et lignes de code source où ces erreurs ont été détectées, ainsi que le nombre de fois qu'elles l'ont été. Cette liste donne donc une indication de la fréquence de chacune des erreurs ainsi que leur emplacement.

Cependant, CUDA ne fournit aucune méthode directe pour accéder aux informations de débogage à partir du programme exécuté. Le seul moyen d'obtenir une trace est d'utiliser le débogueur. Il n'est également pas utile d'insérer les numéros de ligne dans chaque appel à la fonction `enregistrerErreur` dans le code source étant donné qu'au moment de la compilation, le seul numéro de ligne connu est celui de l'appel à partir des opérateurs redéfinis et non celui de l'appel à l'opérateur par le programme original – qui serait l'information pertinente pour localiser un problème. Par ailleurs, insérer les numéros de ligne de chaque appel aux opérateurs requerrait des modifications importantes au code source, ce qui risquerait d'introduire des erreurs et rendrait le module DSA inapplicable à d'autres programmes.

La solution adoptée consiste à insérer les numéros de ligne et les noms de fichier dans le code PTX, le code assembleur commun à tous les GPU de NVIDIA. Le compilateur `nvcc` inclut déjà les numéros de ligne de code source correspondant à chaque série d'instructions PTX, ce qui facilite cette tâche. Pour que cette méthode fonctionne, toutes les fonctions du module DSA remplaçant des opérateurs doivent être *inline*. Ainsi, le code de chaque opérateur est directement inséré dans le code source du programme, ce qui permet d'associer chaque appel avec la ligne appropriée. Inversement, la fonction qui enregistre les erreurs ne doit pas être inline, pour qu'elle apparaisse dans le code PTX comme un appel séparé pour chaque opérateur, ce qui permet d'introduire les numéros de ligne et de fichier à chaque fois.

Le processus de compilation a donc été modifié pour adopter cette nouvelle méthode. La compilation doit être séparée en deux phases. Lors de la première phase, le fichier PTX initial est généré par le compilateur à partir de la portion du code source contenant des appels CUDA. Ensuite, un script traite ce fichier en détectant chaque appel à `enregistrerErreur` et en insérant dans les paramètres les numéros de ligne et de fichier qui y correspondent. La compilation est alors reprise à partir du fichier PTX modifié pour former le code objet, qui est ensuite lié avec les autres objets pour former l'exécutable.

Cette méthode n'est pas conforme à l'utilisation normale du compilateur `nvcc`, qui produit le code objet à partir du code source en un seul appel. Il faut donc recourir à une solution détournée qui fait appel à deux options de `nvcc`. La première (`-keep`) conserve tous les fichiers

intermédiaires générés par une compilation en code objet. La seconde (`-verbose`) affiche les commandes intermédiaires exécutées par `nvcc`, qui est en réalité un enveloppeur pour divers programmes. En utilisant ces commandes directement, à partir du point où le fichier PTX est généré, il est possible de reprendre le processus de compilation avec le nouveau fichier PTX comme s'il n'avait pas été interrompu.

La méthode de détection décrite ci-dessus a été testée à l'aide du même problème que celui décrit à la section 5.1.2. La liste des erreurs détectées a été comparée aux résultats obtenus lors des autres tests afin de déterminer l'efficacité et l'utilité du module DSA pour la détection automatique d'erreurs causées par l'arithmétique flottante.

CHAPITRE 6

RÉSULTATS THÉORIQUES ET EXPÉRIMENTAUX

Ce chapitre présente l'ensemble des résultats pour les expériences décrites au chapitre précédent. La première section se rapporte aux tests reliés à l'article du chapitre 4 ; elle ajoute des détails sur l'accumulation d'énergie dans les voxels, l'injection d'erreur et la précision du résultat de certaines fonctions géométriques. La deuxième section illustre les effets de la taille des voxels sur les résultats, tandis que la troisième présente un survol des résultats obtenus par l'application de la méthode de détection automatique décrite.

La plupart des résultats sont présentés sous forme de comparaison voxel par voxel entre deux distributions de dose. Pour chaque comparaison, seul un sous-ensemble des voxels contenant une dose plus importante a été utilisé. Le seuil de dose utilisé pour que les voxels soient sélectionnés pour la comparaison est de 100 fois la moyenne de dose de tous les voxels, ce qui correspond à environ 0.15% des voxels contenant une dose non nulle. Dans le cas étudié, ce seuil est équivalent à environ 1% de la dose maximale retrouvée dans chaque distribution. Cependant, un seuil proportionnel à la dose moyenne plutôt qu'à la dose maximale est préférable pour obtenir des résultats plus cohérents, comme il sera expliqué plus loin.

Par ailleurs, les comparaisons sont toutes faites entre des distributions calculées avec la même valeur initiale pour la génération de nombres aléatoires, afin d'examiner uniquement l'erreur causée par des effets numériques plutôt que par la nature stochastique de la simulation. Chaque résultat présenté correspond à la moyenne de 8 comparaisons et les barres d'erreurs montrent la taille d'un écart-type.

6.1 Résultats présentés dans l'article

6.1.1 Accumulation de l'énergie

La figure 6.1 présente les résultats des tests de comparaison entre les calculs en précision simple ou double pour l'accumulation de l'énergie seulement, ainsi que quelques résultats omis de l'article au chapitre 4. Pour chaque comparaison effectuée, le pourcentage de différence de dose d'une distribution par rapport à l'autre a été calculé, comme défini à la section 3.2.1, selon l'équation 3.1. La distribution utilisée comme référence a peu d'importance, étant donné que la valeur absolue des différences entre voxels est prise et que la somme de toutes les valeurs de dose dans une distribution varie très peu d'une simulation à une autre.

La première colonne de la figure 6.1 montre la différence entre une simulation lors de

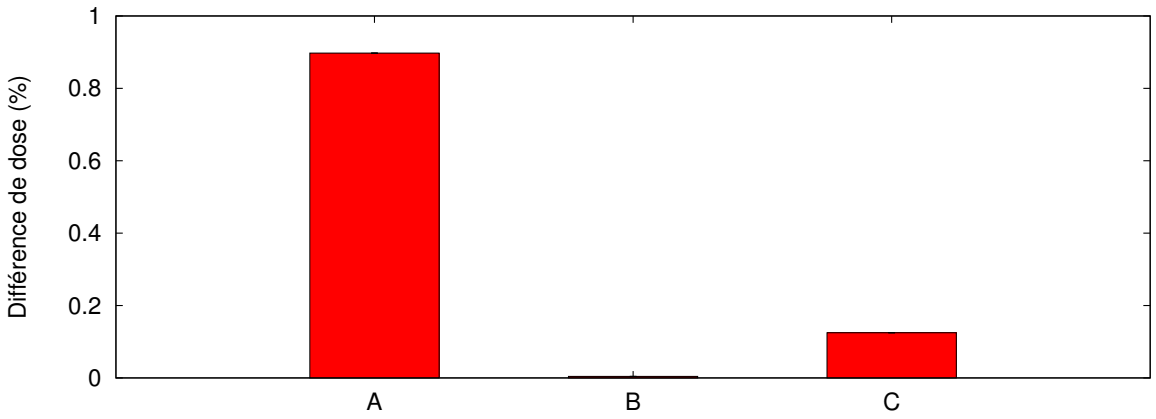

Figure 6.1 Tests supplémentaires sur l'accumulation de l'énergie.A) Différence entre précision simple et précision double. B) Différence entre précision double et accumulation par étapes (précision simple). C) Différence entre précision simple et précision double tronquée.

laquelle l'accumulation de l'énergie est faite en précision simple et une autre où ce calcul est fait en précision double. La deuxième montre la différence observée entre une simulation en précision double et une où l'accumulation se fait par étapes ; l'implantation de cette méthode pour effectuer l'accumulation devait corriger l'erreur causée par le manque de précision lorsque le calcul est fait en précision simple, sans devoir augmenter la précision de ce calcul. La dernière colonne montre la différence entre une simulation en précision simple et une où l'accumulation en précision double a été tronquée selon la méthode décrite plus tôt (section 5.1.2). Le problème utilisé pour ces tests était le même que le premier cas test (section 4.3.1), mais seulement 10^8 photons étaient simulés.

À la suite de la correction de l'erreur lors de l'accumulation de l'énergie, l'effet de la taille des groupes de particules (le nombre de particules par noyau CUDA) a été étudié. La figure 6.2 représente graphiquement le pourcentage de différence de dose observé entre l'accumulation en précision simple et double en fonction du nombre de particules par noyau. Le problème était le même que précédemment, avec 1.92×10^9 photons, afin que le nombre de particules corresponde à un nombre entier de noyaux pour chaque taille testée.

6.1.2 Injection d'erreur

Les tableaux 6.1 à 6.4 présentent les résultats bruts obtenus lors de chaque test d'injection d'erreur effectué. Dans chaque cas, le pourcentage de différence de dose par rapport à une distribution référence (sans erreur) est montré pour les différentes tailles d'erreur insérée. Le problème utilisé pour ces tests est le deuxième cas test décrit à la section 4.3.1, avec 2×10^9

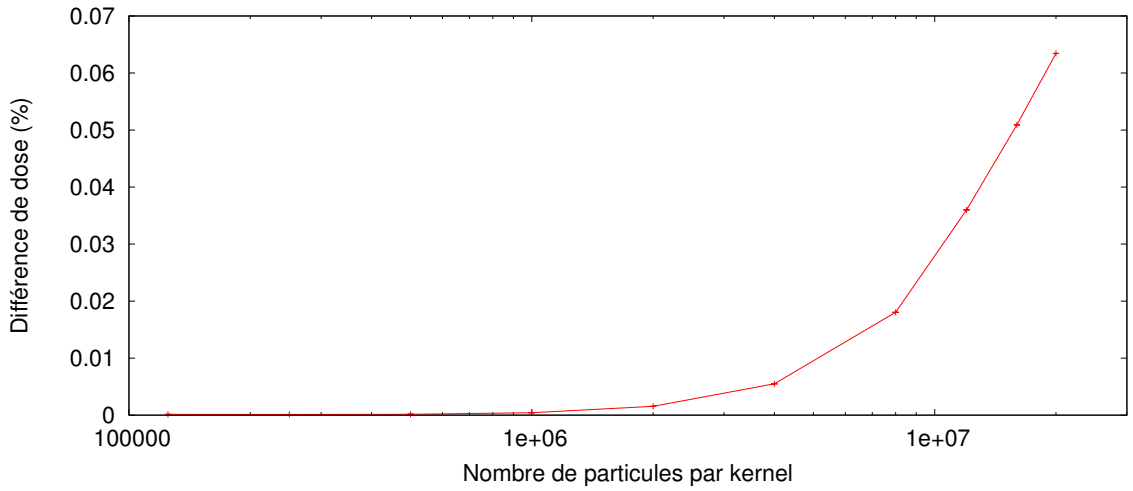

Figure 6.2 Variation entre deux simulations en fonction du nombre de particules par noyau, lorsque l'accumulation d'énergie se fait par étape (précision simple).

photons. À titre de référence, la différence de dose entre deux simulations de ce cas sans erreur introduite, avec une graine différente pour la génération de nombres aléatoires, avec le même nombre de particules, est en moyenne de 0.425%, avec un écart-type de 0.003%.

Dans le tableau 6.1, les erreurs ont été sélectionnées à intervalles réguliers allant de 10^{-6} à 10^{-1} fois le côté d'un voxel (qui était de 0.1 cm^3 lors de ces tests). Pour le tableau 6.2, les intervalles étaient les mêmes, mais constituaient des fractions de cercle (1 unité = 2π radians). Pour le tableau 6.3, les mêmes intervalles étaient utilisés, mais constituaient une fraction sans unité.

Dans le tableau 6.4, les altérations des bits constituent plutôt une erreur relative. Une erreur sur le dernier bit d'un nombre de précision simple correspond à une erreur relative entre 2^{-23} et 2^{-24}. Les erreurs sélectionnées pour ces tests varient donc entre 2^{-16} et 2^{-3}.

Pour les erreurs aléatoires, la taille indiquée correspond à l'amplitude maximale de l'erreur, qui peut être positive ou négative et qui a été sélectionnée selon une distribution uniforme entre ces deux bornes. À l'opposé, l'erreur fixe appliquée est toujours positive et d'amplitude égale au nombre indiqué.

Les figures 6.3 et 6.4 résument les résultats des tableaux 6.1 à 6.4 en illustrant la taille minimale à partir de laquelle les erreurs ont été détectées lors des comparaisons avec une distribution référence. Le seuil utilisé pour considérer une erreur comme détectable est de 5×10^{-3}. À ce seuil, environ 1 voxel sur 10^4 contient une erreur entre 0.05% et 0.1%, qui est la plus petite erreur montrée sur les histogrammes présentés au chapitre 4.

Tableau 6.1 Différence de dose (en pourcentage) par rapport à une exécution de `bGPUMCD` sans erreur ajoutée. L'erreur ajoutée correspond à une fraction de voxel.

Test	Logarithme de la taille de l'erreur ajoutée					
	-6	-5	-4	-3	-2	-1
Index voxel						
Aléatoire	5.62×10^{-5}	6.07×10^{-5}	2.86×10^{-4}	1.39×10^{-3}	1.24×10^{-2}	7.03×10^{-1}
Fixe	1.03×10^{-6}	1.50×10^{-6}	8.41×10^{-6}	3.23×10^{-4}	1.14×10^{-2}	4.86×10^{-1}
Position photon						
Aléatoire	7.71×10^{-3}	3.92×10^{-2}	3.08×10^{-1}	5.65×10^{-1}	3.67×10^{0}	1.13×10^{1}
Fixe	1.10×10^{-2}	1.01×10^{-1}	3.70×10^{-1}	1.25×10^{0}	1.00×10^{1}	2.65×10^{1}
calculeIntersection						
Aléatoire	5.16×10^{-3}	1.43×10^{-2}	3.21×10^{-1}	4.00×10^{-1}	3.84×10^{0}	1.82×10^{1}
Fixe	7.98×10^{-3}	1.62×10^{-2}	9.97×10^{-2}	9.45×10^{-1}	2.20×10^{0}	1.35×10^{1}
calculePas						
Aléatoire	3.36×10^{-3}	8.20×10^{-3}	2.51×10^{-2}	9.09×10^{-2}	3.10×10^{-1}	4.18×10^{0}
Fixe	4.39×10^{-3}	1.05×10^{-2}	2.85×10^{-2}	9.58×10^{-2}	2.93×10^{-1}	4.04×10^{0}
calculeSegment						
Aléatoire	6.37×10^{-6}	1.26×10^{-5}	3.58×10^{-5}	2.10×10^{-4}	2.37×10^{-3}	8.27×10^{-2}
Fixe	9.40×10^{-5}	9.48×10^{-4}	9.50×10^{-3}	9.57×10^{-2}	1.01×10^{0}	1.17×10^{1}

Tableau 6.2 Différence de dose (en pourcentage) par rapport à une exécution de `bGPUMCD` sans erreur ajoutée à la direction des photons. L'erreur ajoutée correspond à une fraction de cercle.

Test	Logarithme de la taille de l'erreur ajoutée					
	-6	-5	-4	-3	-2	-1
Direction photon						
Aléatoire	4.16×10^{-3}	6.17×10^{-3}	1.50×10^{-2}	5.29×10^{-2}	1.98×10^{-1}	4.30×10^{0}
Fixe	4.52×10^{-3}	8.63×10^{-3}	4.25×10^{-2}	3.73×10^{-1}	3.63×10^{0}	2.92×10^{1}

Tableau 6.3 Différence de dose (en pourcentage) par rapport à une exécution de `bGPUMCD` sans erreur ajoutée dans la diffusion de Compton. L'erreur ajoutée correspond à une fraction, additionnée à une valeur entre 0 et 1.

Test	Logarithme de la taille de l'erreur					
	-6	-5	-4	-3	-2	-1
Diffusion Compton						
Aléatoire	2.31×10^{-3}	6.70×10^{-3}	2.24×10^{-2}	8.18×10^{-2}	4.63×10^{-1}	4.82×10^{0}
Fixe	3.20×10^{-3}	9.75×10^{-3}	4.42×10^{-2}	3.49×10^{-1}	3.13×10^{0}	8.77×10^{0}

Tableau 6.4 Différence de dose (en pourcentage) par rapport à une exécution de `bGPUMCD` sans erreur ajoutée. L'erreur est quantifiée en nombre de bits altérés sur la valeur modifiée.

Test	Taille de l'erreur (nombre de bits)					
	8	12	14	16	18	20
Effet photoélectrique (énergies)						
Aléatoire	5.62×10^{-4}	2.55×10^{-3}	6.45×10^{-3}	1.31×10^{-1}	1.89×10^{-1}	1.17×10^{0}
Fixe	9.15×10^{-4}	5.79×10^{-3}	2.26×10^{-2}	2.91×10^{-1}	2.85×10^{-1}	1.09×10^{0}
Effet photoélectrique (seuils)						
Aléatoire	1.75×10^{-3}	8.65×10^{-3}	2.56×10^{-2}	9.37×10^{-2}	3.71×10^{-1}	1.47×10^{0}
Fixe	2.07×10^{-3}	1.07×10^{-2}	3.24×10^{-2}	1.18×10^{-1}	4.67×10^{-1}	1.87×10^{0}
Densités						
Aléatoire	8.97×10^{-4}	2.64×10^{-3}	5.56×10^{-3}	1.48×10^{-2}	5.24×10^{-2}	2.08×10^{-1}
Fixe	1.54×10^{-3}	6.08×10^{-3}	1.68×10^{-2}	6.17×10^{-2}	2.46×10^{-1}	9.84×10^{-1}
Matériaux						
Aléatoire	2.15×10^{-3}	1.26×10^{-2}	4.72×10^{-2}	1.87×10^{-1}	7.46×10^{-1}	2.95×10^{0}
Fixe	3.10×10^{-3}	1.55×10^{-2}	5.15×10^{-2}	1.96×10^{-1}	7.70×10^{-1}	2.91×10^{0}
Facteurs de diffusion						
Aléatoire	6.77×10^{-4}	3.18×10^{-3}	6.39×10^{-3}	1.00×10^{-2}	1.00×10^{-2}	1.00×10^{-2}
Fixe	1.79×10^{-6}	1.05×10^{-6}	1.04×10^{-6}	2.14×10^{-6}	1.04×10^{-6}	6.24×10^{-6}

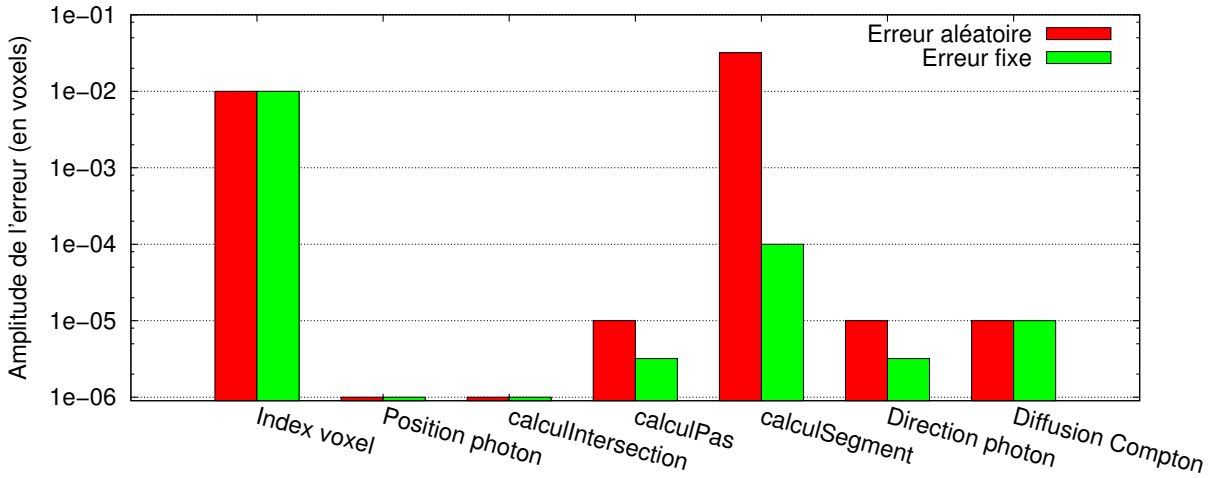

Figure 6.3 Amplitude à partir de laquelle une erreur est détectée dans les diverses parties de `bGPUMCD`.

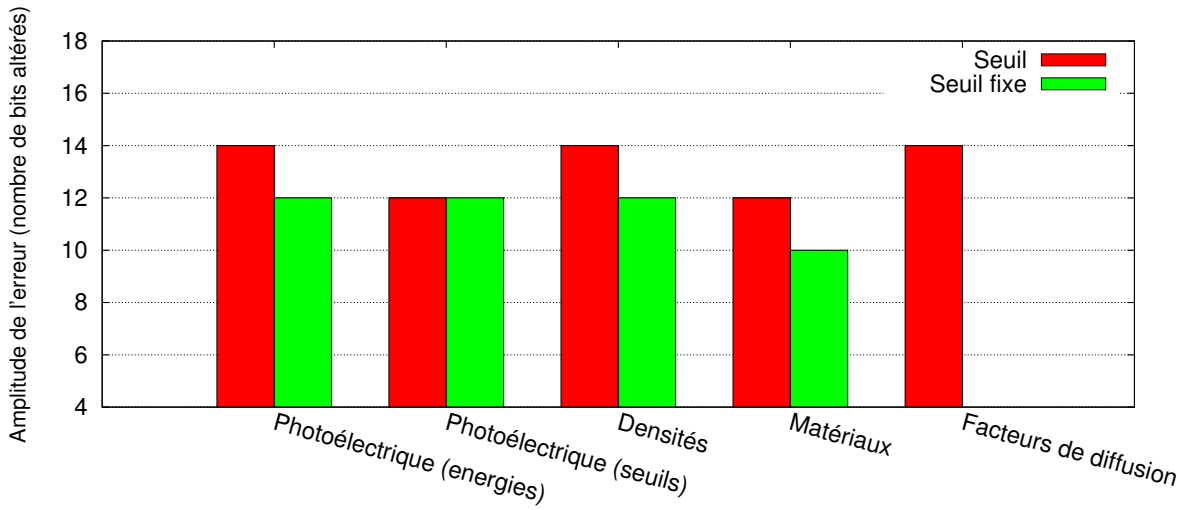

Figure 6.4 Amplitude à partir de laquelle une erreur est détectée dans les diverses parties de `bGPUMCD`.

6.1.3 Fonctions géométriques

La figure 6.5 reprend les résultats présentés dans la figure 4.6 pour la fonction `calculeIntersection` et y ajoute les résultats de tests similaires pour les fonctions `calculeSegment` et `rotateDirectionVector`.

Dans chaque cas, 5 millions d'échantillons ont été sélectionnés uniformément lors d'exécutions de `bGPUMCD`. Chaque échantillon comprend les paramètres passés à la fonction utilisés pour calculer la sortie, ainsi que le résultat final des calculs. Les mêmes paramètres ont été lus pour effectuer le même calcul, avec une précision de 100 décimales (336 bits, soit environ 14 fois plus que la précision simple de la norme IEEE), afin de comparer ce résultat plus précis par celui calculé sur GPU. Les échantillons ont été recueillis par groupes de 50000 par exécution du programme, en raison des limitations de la mémoire globale de la carte graphique utilisée. Ils ont été sélectionnés de façon uniforme tout au long de chaque exécution, afin d'éviter des biais causés par une distribution temporelle non uniforme de l'erreur au cours de l'évolution du système simulé. Par exemple, dans le cas particulier de `calculeIntersection`, toutes les particules s'éloignent des sources au début d'une simulation, ce qui réduit significativement le risque d'erreur lors des calculs d'intersection avec ces sources.

6.2 Discrétisation

Le tableau 6.5 présente les résultats obtenus pour le cas où les voxels ont la même résolution le long des trois axes. Chaque comparaison entre deux distributions a été effectuée deux

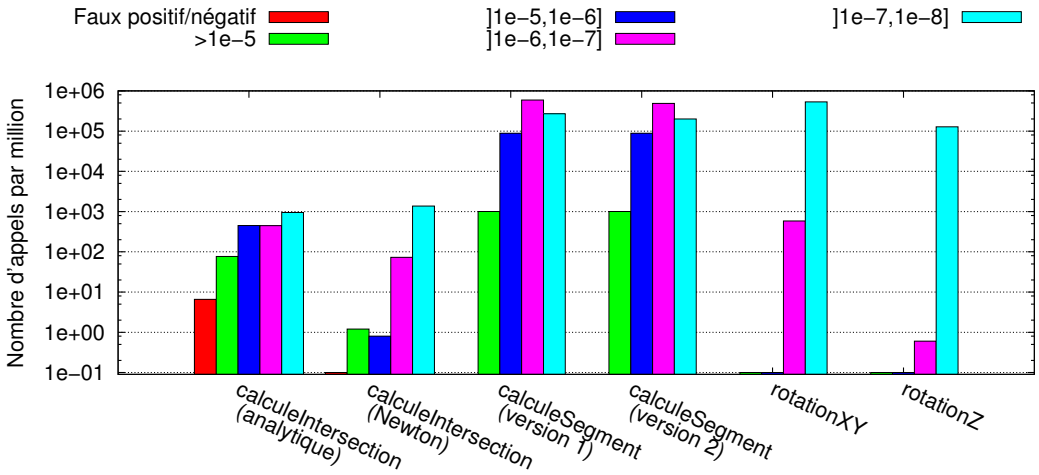

Figure 6.5 Taille des erreurs dans la sortie de différentes fonctions géométriques, sur un million d'appels.

fois, en inversant l'ordre des distributions d'une fois à l'autre. Étant donné que l'algorithme utilisé pour effectuer les comparaisons est différent que pour les résultats déjà présentés, le résultat des comparaisons peut varier selon la distribution utilisée comme référence. La première rangée du tableau indique la résolution de la distribution référence, tandis que la première colonne indique la résolution de la distribution comparée. Dans tous les tests présentés précédemment, la résolution était de $300 \times 300 \times 300$ voxels (pour un cube de 30 cm de côté). Ici, elle varie de $100 \times 100 \times 100$ à $450 \times 450 \times 450$, qui est le maximum possible en raison de la mémoire disponible sur la carte graphique utilisée.

Tableau 6.5 Différence de dose (%) entre les différentes paires de résolutions testées. Le nombre utilisé pour la résolution correspond au nombre de voxels le long de chaque axe.

Distribution	Distribution référence							
évaluée	100	150	200	250	300	350	400	450
100-100-100	–	18.55	26.45	22.30	27.94	24.09	28.59	25.22
150-150-150	5.28	–	9.54	12.43	19.98	14.38	15.83	20.46
200-200-200	0.002	4.24	–	6.57	10.60	10.41	15.97	11.07
250-250-250	1.88	3.26	2.45	–	4.80	6.40	7.70	8.90
300-300-300	0.002	0.001	2.96	1.62	–	3.04	5.32	7.28
350-350-350	0.96	1.61	1.27	1.91	1.66	–	2.57	4.07
400-400-400	0.002	1.38	0.001	1.40	2.26	1.25	–	2.23
450-450-450	0.58	0.002	0.74	0.93	2.04	1.37	0.83	–

Le tableau 6.6 présente les résultats obtenus lorsque la résolution est augmentée le long

d'un axe par rapport à la résolution de base de $300 \times 300 \times 300$.

Tableau 6.6 Différence de dose (%) entre les paires de distributions dont la résolution varie le long de chaque axe. Les nombres utilisés pour indiquer la résolution correspondent au nombre de voxels le long de chaque axe ($x/y/z$).

Distribution évaluée	Distribution référence 300/300/300	600/300/300	300/600/300	300/300/600
300-300-300	–	7.19	7.19	4.20
600-300-300	0.001	–	7.17	4.19
300-600-300	0.001	7.17	–	4.19
300-300-600	0.001	7.17	7.17	–

La figure 6.6 montre le temps de calcul nécessaire par noyau en fonction du nombre de voxels par côté du volume simulé. Pour les tests avec des résolutions différentes selon les axes, le temps obtenu est de 0.6 seconde par noyau lorsque la résolution est doublée en X et en Y et de 0.22 seconde par noyau lorsque la résolution est doublée en Z.

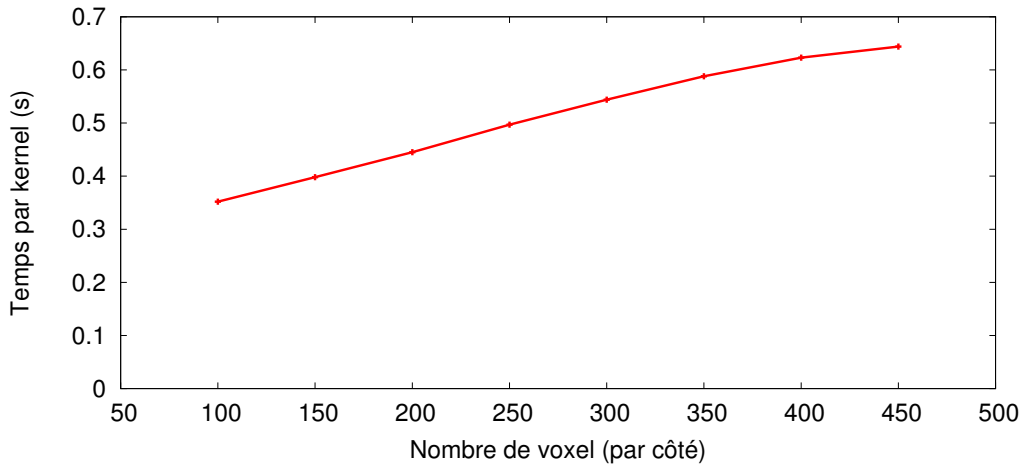

Figure 6.6 Temps de calcul (en secondes) par noyau en fonction du nombre de voxels le long de chaque axe.

6.3 Détection automatique

Les résultats présentés dans cette section sont limités à une énumération des régions de `bGPUMCD` où les événements décrits dans la section 5.4.2 ont été détectés par le module DSA. Ils seront décrits plus en détails dans le prochain chapitre. Le tableau 6.7 présente le nombre d'événements détectés selon la partie du programme où ils se sont produits et

selon l'algorithme utilisé pour la fonction `calculeIntersection` (voir section 4.4.3). Les nombres inscrits sont approximatifs et ont été recueillis au cours d'une exécution pour chaque algorithme, lors d'une simulation du deuxième cas test présenté à la section 4.3.1 (58 sources dans un cube d'eau de 30x30x30 cm, avec des voxels cubiques de 0.1 cm de côté). Cependant, en raison du temps de calcul beaucoup plus long, seulement 10^5 photons ont été simulés, par groupes de 2000 photons. Le facteur de ralentissement causé par l'utilisation du module DSA (par rapport à une exécution normale du programme) est également indiqué.

Tableau 6.7 Nombre d'erreurs détectées par section ou fonction de `bGPUMCD`. *Noyau* fait référence à la fonction utilisée comme point d'entrée du noyau CUDA.

Section	Analytique	Newton
1. `calculeIntersection`	5060000	470000
2. `pointInterieur`	54000	169000
3. `calculeSegment`	46000	47000
4. `calculePas`	21000	15000
5. *Noyau*	1000	1000
Facteur de ralentissement	6400	90000

Lorsque l'algorithme analytique est utilisé, environ 43% des événements détectés au sein de la fonction `distanceRayQuadric` sont une annulation lors du calcul du radical b^2-4ac, à la ligne 4 de l'algorithme 4.1. Cette annulation est suivie d'un embranchement instable lorsque son résultat est comparé avec zéro (25% des événements, ligne 5) et de deux opérations mathématiques instables (28% des événements, lignes 8 et 9). Environ 3.3% des événements sont attribués à une annulation lors du calcul du produit scalaire entre la direction de la particule et la normale de la surface (ligne 2). Environ 0.23% sont attribués au calcul de la distance avec la boîte englobante de la surface quadrique, nécessaire lorsque la particule traverse les extrémités d'un cylindre (non présenté dans l'algorithme 4.1. Enfin, le reste des erreurs (0.51%) n'ont pas été attribuées à une ligne particulière de la fonction.

Lorsque c'est la méthode de Newton qui est utilisée, la grande majorité des événements détectés (93%) sont une annulation lors du calcul de la dérivée du polynôme dont on cherche les racines. Environ 5.5% ne sont pas attribués à une ligne particulière.

CHAPITRE 7

DISCUSSION GÉNÉRALE

7.1 Résultats présentés dans l'article

7.1.1 Accumulation de l'énergie

En considérant l'exécution en double précision comme la plus réaliste, il est clair d'après la figure 6.1 qu'il y a une différence importante entre les deux précisions (colonne A) en ce qui concerne l'accumulation de l'énergie, et que la modification apportée corrige presque entièrement cette différence (colonne B). De plus, la colonne C montre que la différence observée entre les précisions simple et double est en grande partie causée par le phénomène décrit à la section 4.4.1, c'est-à-dire que les contributions individuelles qui sont trop petites par rapport à la dose totale sont ignorées, avec toutefois une nuance. Lorsque l'écart entre les nombres additionnés est de 2^{24} à 2^{25} fois la taille du plus petit nombre, le plus grand est incrémenté d'une ULP (un nombre dont la valeur est égale à celle du dernier bit est ajouté au total).

Cet effet peut être expliqué par la manière dont l'opération d'addition doit être réalisée lorsqu'un bit "de garde" est utilisé. Même si le coefficient d'un nombre en précision simple contient seulement 23 bits, il est implicitement précédé d'un bit dont la valeur est 1. La seule exception concerne les nombres dénormalisés, dont le bit implicite a une valeur de 0. Cependant, les contributions au total de la dose ne comprennent aucun nombre dénormalisé. Le bit de garde ajoute un bit lors du calcul de l'addition, qui permet de limiter grandement la perte de précision. L'équation 7.1 illustre l'addition de deux nombres dont les exposants ont une différence de 25. La barre verticale indique la séparation entre le 24^e et le 25^e bit.

$$\begin{aligned} & 1.001...011|00000... \\ & \underline{+0.000...000|01101...} \\ & 1.001...011|1 \\ & \widehat{=} 1.001...100| \end{aligned} \tag{7.1}$$

Le 25^e bit du second nombre est arrondi à 1, puis est ajouté au premier nombre. En arrondissant ce résultat au nombre pair le plus près, on obtient le 4^e nombre indiqué, qui est l'équivalent d'une addition de 1 sur le 24^e bit du nombre initial. Si la différence entre les exposants était supérieure à 25, tous les bits du plus petit nombre seraient zéros après l'ali-

gnement des coefficients et l'arrondi. À l'inverse, lorsque la différence est inférieure à 25, le petit nombre est utilisé, mais grandement arrondi. Étant donné que cet arrondi est fait en proportions approximativement égales vers le haut et vers le bas – en raison du caractère aléatoire de la simulation – l'erreur qu'il cause s'annule en grande partie. Cependant, comme il n'a pas été simulé lors des tests avec l'addition en double précision modifiée, il pourrait être à l'origine des différences observées entre la sommation en simple précision et celle en double précision modifiée dans la colonne C de la figure 6.1.

Les résultats présentés à la figure 6.2 indiquent que le nombre de photons par kernel commence à affecter le résultat final d'une simulation lorsqu'il atteint environ 1 million. Lorsqu'une seule source est présente dans l'environnement simulé, environ la moitié des photons passent à travers chacun des quelques voxels qui contiennent la source. Vers la fin de la simulation, les contributions individuelles des particules sont environ 2^{20} fois plus petites que la dose totale dans ces voxels en moyenne (pour 1 million de photons). Étant donné qu'il existe une grande variabilité dans les tailles de ces contributions, en raison des différences de densité et de la distance parcourue au sein d'un voxel, une partie de ces contributions sont trop petites et sont ignorées lors de l'addition. Plus le nombre de particules simulées par kernel augmente, plus la proportion des contributions ignorées augmente dans ces voxels et plus les voxels voisins sont aussi affectés par ce phénomène, ce qui explique la tendance vers le haut de la courbe après un million de particules.

Initialement, la correction du problème d'accumulation de l'énergie a causé une augmentation du temps d'exécution d'environ 1% pour chaque kernel (au moment d'écrire l'article en section 4). Cependant, les kernels contenaient 10 millions de particules chacun. En divisant par 10 la taille des kernels, pour obtenir une exactitude maximale des résultats, le nombre de kernels à exécuter pour simuler le même nombre de photons est multiplié par 10, tout comme le coût de la correction. L'augmentation du temps calcul devient alors beaucoup plus importante, ce qui rend souhaitable le développement d'une nouvelle solution.

Une variante de cette solution facile à implanter serait d'utiliser deux (ou plusieurs) tampons pour contenir les résultats. Ainsi, les contributions des particules seraient distribuées à travers plusieurs ensembles de résultats, ce qui permettrait de simuler plus de particules par kernel, divisant ainsi le coût de la solution par le nombre de tampons utilisés.

7.1.2 Injection d'erreur

Pour les besoins de cette discussion, les résultats présentés à la section 6.1.2 seront divisés en trois parties : la première concerne les différents calculs de distance utilisés pour déterminer le déplacement des particules, la deuxième comprend les variations ajoutées directement sur la position et la direction des particules – qui influencent à peu près tous les calculs dans

bGPUMCD – et la troisième inclut les trois types d'interaction simulés, ainsi que les données physiques qui déterminent les probabilités des événements et leurs effets sur l'énergie et la position des particules.

Mouvement des particules

Trois éléments du mouvement des particules ont été testés en y ajoutant des erreurs. La fonction `calculeIntersection`, décrite en détail à la section 4.4.3, sert à calculer la distance entre une particule et la surface des divers éléments des sources de radiation. Elle est utilisée pour calculer la distance entre deux interactions pour une particule (ou "pas"). Elle est appelée de façon itérative par `calculePas`, qui détermine chaque fois la distance parcourue par une particule dans un milieu d'une certaine densité et utilise cette distance, ainsi que le libre parcours de la particule, pour connaître la distance physique totale parcoure par cette particule avant d'être impliquée dans une interaction. Enfin, la distance calculée par `calculePas` est utilisée, à l'aide de `calculeSegment`, pour calculer la longueur du parcours de la particule à travers chacun des voxels le long de cette trajectoire et connaître ainsi la quantité d'énergie déposée dans chacun de ces voxels.

L'importance de calculer précisément ces distances varie grandement selon chacune des fonctions, d'après les résultats du tableau 6.1 et de la figure 6.3. En effet, une erreur aléatoire de moins de 10^{-6} voxel dans la valeur retournée par `calculeIntersection` se remarque dans le résultat final d'une simulation, tandis que l'erreur doit être environ 10 fois plus grande dans le résultat de `calculePas` et 10^4 fois plus grande dans celui de `calculeSegment` pour être remarquée dans la distribution de dose.

Pour `calculeSegment`, la grande tolérance aux erreurs aléatoires ainsi que la faible tolérance aux erreurs fixes (environ 300 fois moins grande) corroborent les résultats présentés plus tôt qui suggèrent que les erreurs aléatoires ont tendance à s'annuler lors de la sommation qui accumule la dose dans chaque voxel. De plus, cette tolérance est amplifiée dans les voxels où passent beaucoup de particules – ceux qui comptent d'ailleurs le plus lors des comparaisons entre distributions de dose : après qu'environ 5000 photons aient traversé un voxel, plus de la moitié des chiffres significatifs des contributions ultérieures sont perdus lors de l'addition avec l'énergie totale en précision simple.

Inversement, la fonction `calculePas` est approximativement aussi sensible aux erreurs aléatoires qu'aux erreurs fixes. Bien qu'aucune explication ne soit évidente pour ce résultat quelque peu surprenant (l'amplitude de l'ensemble des erreurs fixes est environ deux fois plus grande que celle des erreurs aléatoires), il est possible que les effets beaucoup plus complexes de cette fonction, par rapport à ceux de `calculeSegment`, en soient la cause. En effet, la distance entre deux interactions est directement responsable de la position de la particule lors

de la prochaine itération de l'algorithme ; une erreur pourrait donc être amplifiée à partir de ce point en affectant les calculs ultérieurs, pour donner une série de trajectoires devenues invalides. La fonction `calculeSegment`, au contraire, n'affecte aucunement les calculs suivants. Par ailleurs, l'erreur dans le résultat de `calculePas` a peu d'effet sur le calcul des distances dans chaque voxel, donc des contributions individuelles, étant donné qu'elle n'aurait d'effet que sur le dernier voxel sur la trajectoire et que les erreurs n'ont un effet sur l'accumulation qu'à partir d'une amplitude beaucoup plus grande.

En ce qui concerne `calculeIntersection`, elle n'est appelée que par `calculePas`. Toute erreur qu'elle génère doit donc nécessairement se transmettre à travers cette dernière. La différence entre les seuils d'erreur détectables des deux fonctions indique que `calculePas` amplifie l'erreur de `calculeIntersection`. Ce résultat est normal dans les circonstances du test effectué, mais ne reflète pas adéquatement ce qui se produit en réalité, lorsqu'aucune erreur n'est ajoutée aux résultats des calculs.

Deux cas d'erreur sont possibles : soit la valeur retournée par `calculeIntersection` est en dessous de la valeur théorique, soit elle est au-dessus. Lorsque la valeur est trop faible, il ne devrait y avoir aucun effet autre que d'augmenter légèrement le temps de calcul. Le prochain appel par `calculePas` devrait en effet tout simplement trouver la nouvelle distance avec la même surface et ajouter cette distance au parcours de la particule. Par contre, lorsque la valeur retournée est trop élevée, la particule traverse deux milieux différents d'un seul coup. Dans cette situation, `calculePas` ne tient compte que d'un seul de ces milieux pour mettre à jour le libre parcours de la particule, qui devient alors trop court ou trop long, dépendamment de la densité relative des deux milieux. Étant donné que l'épaisseur d'une couche de matériau dans la source *SelectSeed* (voir section 2.2) simulée durant les tests peut être aussi mince que 0.003 mm, même une erreur très faible peut constituer une fraction significative de l'épaisseur du matériau.

Lors des tests effectués, une erreur était ajoutée à chaque résultat de `calculeIntersection`. Ainsi, lorsque l'erreur était négative, elle n'avait aucun effet autre que d'ajouter une itération au calcul, alors que si elle était positive elle modifiait également le libre parcours de la particule. Ce libre parcours est alors raccourci ou allongé, selon le matériau. Le matériau sélectionné est celui dans lequel la particule se trouve à mi-chemin de la distance physique qu'elle parcourt. Comme l'erreur sur le libre parcours varie à chaque itération, elle s'annule en partie. Cet effet explique que la différence observée dans le résultat final est parfois plus élevée et parfois moins entre l'erreur aléatoire et l'erreur fixe pour la fonction `distanceRayQuadric`.

Positionnement des particules

Trois aspects liés au positionnement des particules ont été testés : l'identification du voxel dans lequel les photons se trouvent, leur position et leur direction. Le premier élément est directement relié à la position ainsi qu'à la distance parcourue par les photons entre deux interactions. La simulation semble tolérer des erreurs relativement grandes dans la position lorsque celle-ci sert à déterminer le voxel dans lequel une particule se situe. Il est cependant important de noter que les erreurs fixes y ont moins d'impact que les erreurs aléatoires, ce qui indique une annulation probable de certaines erreurs. En effet, l'association d'un voxel avec une particule sert principalement à déterminer à travers combien de voxels elle passe, dans le but de calculer la distance parcourue à l'intérieur de chacun d'eux, à l'aide de la fonction `distanceInVoxel`. Par contre, le dernier segment de la trajectoire avant l'interaction est calculé à partir de la somme de tous les autres, ce qui engendre une irrégularité parmi les erreurs.

Quatre scénarios possibles, selon que l'erreur ajoute ou enlève des voxels au trajet de la particule et selon que le voxel final se trouve sur la trajectoire ou non, sont illustrés (en deux dimensions) par la figure 7.1 et seront décrits ici. L'erreur peut impliquer plusieurs voxels étant donné qu'elle est ajoutée sur les trois axes, avec des valeurs différentes. Pour chaque axe, la distance est calculée pour chaque frontière de voxel traversée le long de cet axe. La dose déposée dans le voxel final est calculée à partir de la distance totale entre les deux interactions et la distance parcourue dans chaque voxel précédent.

A) *Un voxel est soustrait et le voxel final* ***n'est pas*** *sur la trajectoire de la particule.* Trois voxels contiennent une valeur erronée : le voxel final correct, qui ne reçoit aucune énergie de la particule, le dernier voxel le long de l'axe dont il a été soustrait, qui ne reçoit également rien, et le nouveau voxel final, qui reçoit la dose qui aurait dû être attribuée aux deux autres. La dose totale déposée lors de la simulation n'est aucunement changée ; l'erreur en différence de dose est proportionnelle à environ 2 fois la valeur de l'erreur sur la position plus 2 fois la longueur du segment transféré d'un voxel à l'autre.

B) *Un voxel est soustrait et le voxel final* ***est*** *sur la trajectoire de la particule.* Deux voxels se retrouvent avec une valeur erronée : le voxel final correct, qui ne reçoit rien, et le dernier voxel le long de l'axe dont il a été soustrait (également le nouveau voxel final), auquel est ajoutée en plus de sa dose correcte la dose qu'aurait dû recevoir le vrai voxel final. La dose totale déposée lors de la simulation n'est aucunement changée ; l'erreur en différence de dose est proportionnelle à environ 2 fois la valeur de l'erreur sur la position.

C) *Un voxel est ajouté et le voxel final* ***n'est pas*** *sur la trajectoire de la particule.* Trois voxels peuvent se retrouver avec une valeur erronée : le voxel final correct, qui ne reçoit rien, un voxel supplémentaire le long de l'axe auquel il a été ajouté, qui reçoit une dose proportionnelle

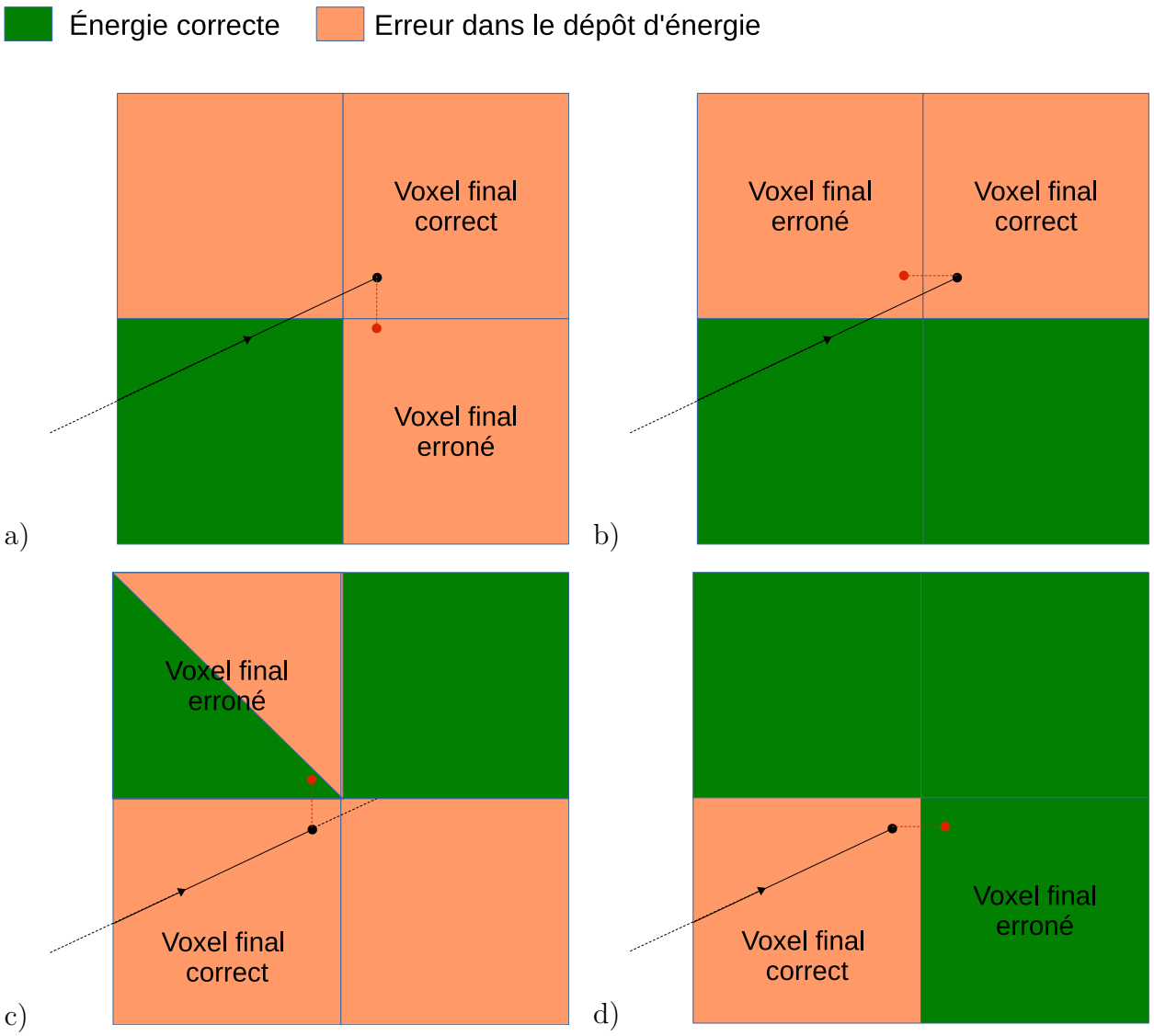

Figure 7.1 Scénarios possibles lorsqu'une erreur sur la position d'une particule l'associe avec le mauvais voxel. a) Voxel enlevé, nouveau voxel final en dehors de la trajectoire. b) Voxel enlevé, nouveau voxel final sur la trajectoire. c) Voxel ajouté, nouveau voxel final en dehors de la trajectoire. d) Voxel ajouté, nouveau voxel final sur la trajectoire.

à la longueur du segment de la trajectoire projetée à travers lui, et le nouveau voxel final, qui reçoit le restant de la distance entre les deux interactions. Si ce restant est négatif, il ne reçoit rien du tout ; la dose totale déposée lors de la simulation est alors supérieure à ce qu'elle devrait être. L'erreur en différence de dose est proportionnelle à environ 1 fois la valeur de l'erreur sur la position.

D) *Un voxel est ajouté et le voxel final* ***est*** *sur la trajectoire de la particule.* Un seul voxel se retrouve alors avec une valeur erronée : le voxel final correct, qui reçoit une dose proportionnelle à la longueur du segment de la trajectoire projetée à travers lui (et qui est supérieure à la dose correcte). Le nouveau voxel final ne reçoit rien étant donné que la distance restante à parcourir est négative. La dose totale déposée lors de la simulation est supérieure à ce qu'elle devrait être ; l'erreur en différence de dose est proportionnelle à environ 1 fois la valeur de l'erreur sur la position.

La probabilité d'occurrence n'est pas égale pour tous les scénarios décrits ci-dessus. Il est en effet beaucoup plus probable qu'une erreur situe le nouveau voxel final du parcours d'une particule le long de sa trajectoire (scénarios B et D). L'inverse nécessite que la particule se trouve très près de deux voxels voisins différents.

Lors des tests, une erreur était appliquée sur chacun des trois axes. Ainsi, si ces trois erreurs provoquaient un changement de voxel le long de plus d'un axe à la fois, plusieurs des scénarios décrits ci-dessus se produisaient simultanément. Lors des tests où l'erreur ajoutée est d'amplitude fixe, le scénario A implique que le scénario B se produit également et le scénario C implique que le scénario D se produit également. Par contre, lors des tests où l'erreur ajoutée est aléatoire, dans plus de la moitié des cas où le scénario A se produit, le scénario B est également présent (le même ratio existe pour les scénarios C et D). Dans les deux séries de tests, il est possible que les scénarios B et D se produisent en même temps.

L'impact des scénarios sur les résultats finaux change lorsque plusieurs se produisent en même temps. La combinaison qui est responsable de la différence observée entre les erreurs aléatoires et fixes (tableau 6.1 et figure 6.3) est possiblement celle des scénarios A et B. En effet, dans un tel cas, l'erreur en différence de dose est proportionnelle à environ 4 fois la valeur de l'erreur sur la position, ce qui est généralement beaucoup plus faible de l'erreur du scénario A uniquement. Ainsi, lorsqu'une erreur d'amplitude fixe est ajoutée le long de chaque axe, son effet sur le dépôt de dose s'annule plus fréquemment qu'avec une erreur aléatoire. La différence observée entre les deux séries de tests sur l'assignation d'une position à un voxel est donc causée par la variation de l'erreur plutôt que simplement sa grandeur.

Le deuxième aspect du positionnement testé, soit la position même des photons, est beaucoup plus sensible aux erreurs que l'association des photons avec un voxel. En fait, même la plus petite erreur introduite génère une déviation détectable par rapport à une simulation

sans erreur. L'erreur introduite affecte entre autres le voxel dans lequel la particule se trouve. Cependant, cela n'explique que partiellement la différence de dose observée, puisque l'erreur a un effet même à des niveaux indétectables pour l'association avec un voxel.

Deux phénomènes peuvent aider à expliquer cette sensibilité. Le premier est qu'une petite déviation de la position de la particule, avec une direction différente de celle de la particule, affecte la dose de chaque voxel sur sa trajectoire. Le nombre de voxels affectés – proportionnel à la différence de dose générée – augmente avec la longueur du pas entre deux interactions.

Le second phénomène est plutôt causé par la manière dont l'erreur est ajoutée. En effet, elle est insérée à chaque itération de l'algorithme principal, c'est-à-dire avant chaque calcul de la distance avec la prochaine interaction d'une particule. Ainsi, plus une particule entre en interaction, plus sa position dévie. Les erreurs aléatoires successives sur un photon peuvent parfois diminuer la déviation, mais comme elles sont appliquées sur les trois axes, il est plus probable qu'elles l'augmentent. Les erreurs fixes, quant à elles, ne font qu'augmenter la déviation avec chaque itération, ce qui explique qu'elles aient un effet beaucoup plus grand sur la différence de dose que les erreurs aléatoires.

Il est plus difficile d'expliquer le fait que les erreurs aléatoires parmi les différentes particules ne s'annulent pas. Avec 2×10^9 particules, la distribution de dose contient relativement peu de variations causées par les photons individuels – du moins dans les voxels testés, qui contiennent une dose suffisamment élevée. Il est donc normal de supposer qu'une faible déviation appliquée de façon aléatoire ne se remarque pas. La différence est possiblement causée par les photons qui subissent le plus d'interactions, étant donné qu'ils dérivent de plus en plus à chaque fois. L'autre possibilité est que ce soient tous les photons à leur origine, alors que la plus petite déviation peut placer les particules dans un milieu différent (à l'intérieur des sources). Ainsi, les interactions qui ont lieu au tout début de la simulation peuvent être changées, ce qui implique un changement dans toutes les interactions suivantes. Plus de tests permettraient de déterminer si une de ces hypothèses est vraie, par exemple en changeant uniquement la position initiale des particules plutôt qu'à chaque itération.

Les tests sur le troisième aspect du positionnement des particules, leur direction, suggèrent que c'est la position initiale qui est la plus importante. Il est difficile de comparer directement les effets de la taille de l'erreur sur la distribution de dose, étant donné que les unités utilisées sont différentes. Cependant, il est possible de calculer l'effet d'un changement de direction sur la position. La plus petite erreur aléatoire détectée est de 10^{-5}, ce qui équivaut à une déviation maximale de 5×10^{-4} degré. Sur une distance d'un centimètre, qui est relativement courte pour une particule dans l'eau, cet angle correspond à environ 10^{-5} cm de déviation, ou 10^{-4} voxel lors des tests. Cette déviation est 10 fois plus importante que la plus petite erreur détectable sur la position. Pourtant, l'effet d'un changement de direction est très similaire

à celui d'un changement de position : tous les voxels sur la trajectoire de la particule sont affectés. De la même façon que l'erreur sur la position, celle sur la direction s'accumule à chaque itération. Par contre, l'erreur aléatoire s'annule plus que l'erreur d'amplitude fixe, qui est détectable à un seuil environ 3 fois inférieur à celui de l'erreur aléatoire. La seule différence survient lors de la première itération : lorsque la direction est modifiée, la position est erronée seulement après que la particule ait été déplacée, ce qui indique que la précision de la position initiale est essentielle pour la simulation. Cette conclusion est appuyée par le fait que les photons sont générés dans une partie spécifique très mince de la source, où le moindre déplacement peut situer une particule dans un milieu différent, avec une probabilité d'interaction très différente.

Une autre conclusion importante de ces tests sur le positionnement des particules est que la subdivision en voxels a peu d'impact sur le résultat d'une simulation, comparée à la position des particules. D'après les résultats obtenus, il est probable qu'un changement dans la taille des voxels n'ait pas d'effet sur une simulation étant donné que la plupart des erreurs s'annulent. Une subdivision plus fine entraînerait une granularité plus fine des erreurs, mais celles-ci auraient le même effet lors d'une comparaison voxel par voxel. Un effet similaire se produirait pour une subdivision plus grossière.

Données physiques et interactions

Les données physiques affectent les probabilités d'interaction, leurs proportions ainsi que leurs effets sur les particules. D'après les résultats présentés à la section 6.1.2, leur précision a un effet très similaire sur une simulation. En effet, pour tous les facteurs testés, une erreur est détectable à partir d'un seuil correspondant à environ 0.1% à 0.025% de la valeur modifiée (les 12 ou 14 derniers bits du nombre, en représentation en simple précision). Il est donc clair que toute erreur dans une simulation qui proviendrait de ces données serait causée par une imprécision dans leur mesure plutôt que dans leur représentation – qui est leur seule source d'erreur numérique dans ce programme.

Comme mentionné à la section 4.4.4, cette faible sensibilité aux erreurs est expliquée par le fait que toutes les données physiques sont utilisées à travers des tables de conversion. Ainsi, un petit changement dans ces données ne cause d'erreur que dans une faible proportion des cas – ceux qui se trouvent très près des limites entre les différentes entrées dans les tables.

Il est à noter que les tests ont été effectués dans un milieu complètement uniforme, comprenant uniquement de l'eau. Il est probable que les différences observées soient sensiblement plus élevées dans un milieu plus hétérogène, surtout au niveau des erreurs sur la densité des matériaux.

Le seul groupe de données qui ressort de ces tests est celui des facteurs de diffusion,

utilisés pour calculer l'effet de la diffusion de Rayleigh. Aucune erreur d'amplitude fixe n'a été détectée. Ce résultat est probablement causé par la structure de la fonction calculant l'effet de la diffusion de Rayleigh. Cette fonction choisit aléatoirement un angle de déviation pour le photon impliqué, puis effectue un test basé sur l'énergie du photon et le facteur de diffusion du milieu pour déterminer si cet angle est correct. Dans la majorité des cas, le test réussit du premier coup, ce qui rend l'angle de déviation essentiellement aléatoire. Une sous-estimation des facteurs de diffusion (qui se produit uniquement avec l'erreur aléatoire) entraîne un rejet plus fréquent de la valeur sélectionnée, ce qui expliquerait la différence observée lorsque les erreurs sont aléatoires.

Pour la diffusion de Compton, les résultats sont plus difficiles à interpréter. Le résultat d'une simulation y semble très sensible. Cependant, la valeur sur laquelle l'erreur est ajoutée, qui se trouve entre 0 et 1, n'est pas uniformément distribuée et le calcul qui y mène fait appel à plusieurs nombres aléatoires. Il n'est donc pas possible de définir exactement d'où viendrait l'erreur introduite, ce qui met en doute la validité de ce test.

L'effet photoélectrique est relativement peu sensible aux changements dans les plateaux d'énergie et les seuils des densités de probabilité qui sélectionnent la quantité d'énergie perdue. Les différences observées sont relativement faibles, en raison probablement de la faible différence entre les valeurs des plateaux d'énergie et de leurs probabilités, qui font que le choix d'un plateau ou d'un autre a peu d'impact sur une particule. De plus, dans bGPUMCD, l'énergie restante d'une particule détermine seulement si elle continue à se déplacer (jusqu'à sa prochaine interaction) et non la distance qui lui reste à parcourir. Ainsi, à moins que l'énergie d'un photon devienne nulle, il demeure dans la simulation, sans le moindre changement dans la dose déposée.

Utilité et limitations

La méthode d'injection d'erreur comporte plusieurs inconvénients, par sa nature même : le réalisme des erreurs ajoutées, le choix de leur taille, le mécanisme de leur application et la validité des résultats qu'elles entraînent, par exemple. Toutes les erreurs ajoutées dans bGPUMCD l'ont été pour une raison et ont été justifiées. Cependant, elles ne représentaient pas nécessairement un résultat réaliste des calculs impliqués, surtout en raison du fait qu'elles étaient ajoutées systématiquement. Une liste des critiques sur les erreurs introduites, qu'elles soient inhérentes à la méthode de test ou à sa mise en pratique dans le cadre de cette étude, est présentée ici :

— **calculeIntersection**. L'erreur ajoutée sur le résultat de la fonction pouvait n'avoir aucun effet (si elle était négative). Cependant, une nouvelle erreur était alors ajoutée, jusqu'à ce qu'elle soit positive. En réalité, dans la majorité des cas, l'erreur ne se

produit qu'une seule fois.

— **calculePas**. L'erreur ajoutée sur le résultat est quelque peu plus réaliste que sur `calculeIntersection`, mais il plus difficile de quantifier ce qui constituerait une erreur possible. Une erreur dans le résultat de cette fonction viendrait surtout de `calculeIntersection`.

— **Index des voxels**. Une erreur dans cette fonction proviendrait en réalité d'une erreur sur la position et aurait donc plus d'impact que lors de ce test particulier.

— **Position et direction des photons**. L'erreur était appliquée constamment, à chaque itération, alors qu'en réalité, le comportement de l'erreur est beaucoup plus erratique.

— **Position des photons**. L'erreur proviendrait du résultat de `calculePas` et devrait donc être appliquée le long de la direction, avec une *très faible* variation le long de chaque axe, plutôt que dans une direction aléatoire.

— **Direction des photons**. L'erreur proviendrait surtout de la fonction qui effectue la rotation et de l'implantation de l'effet Compton. Cependant, la fonction de rotation ne génère pratiquement pas d'erreur (voir figure 6.5) et l'erreur venant de l'effet Compton n'est pas uniforme.

— **Direction des photons**. L'erreur aléatoire n'a pas été appliquée de façon totalement aléatoire. Seule la valeur de la déviation du vecteur était choisie complètement au hasard. La direction de la déviation était choisie à l'intérieur d'un cône très restreint.

Un problème qui survient pour toutes les fonctions mentionnées ci-dessus est que l'erreur est appliquée à chaque appel ou chaque itération, ce qui la rend peu réaliste. Les seules sections qui en sont libres sont les données physiques, qui sont modifiées une seule fois à l'initialisation du programme.

Un autre inconvénient, qui vient cette fois-ci de la méthode utilisée pour effectuer les comparaisons entre distributions de dose, est que l'indice obtenu ne permet aucunement de déterminer d'où provient la différence, à partir du code ou de la distribution résultante même.

Malgré ces problèmes, la méthode d'injection d'erreur a permis de connaître plus en profondeur les fonctions testées et leurs interactions. Elle a également permis l'établissement de certaines conclusions majeures, par rapport à l'importance de la précision de la position des particules, de la discrétisation en voxels et des données physiques – en particulier pour les facteurs de diffusion. De façon plus générale, cette méthode permet d'examiner le comportement de certaines fonctions ou sections de programme sous des conditions extrêmes qui, même si elles ne sont pas réalistes, donnent une indication des limitations des algorithmes utilisés.

7.1.3 Fonctions géométriques

Les résultats de la figure 6.5 complémentent ceux discutés dans la section précédente en donnant une indication de l'erreur numérique réelle générée par certaines fonctions. La conclusion principale est que ces erreurs ne sont pas suffisamment grandes et fréquentes pour être détectées dans le résultat final.

Les tests sur les fonctions géométriques ont par ailleurs permis de découvrir quelques faits d'intérêt pour chacune d'elles. Pour le calcul de la distance entre deux interactions, il existait avant le changement d'algorithme un seuil arbitraire (10^{-6}) sous lequel toute distance était ignorée. Ce seuil servait à régler en partie le problème décrit à la section 4.4.3 qui causait une augmentation du temps d'exécution en répétant de nombreuses fois un calcul qui résultait en une distance trop courte. Il permettait également d'éviter un problème qui causait un arrêt du programme. Des tests supplémentaires ont révélé que si la plus petite distance retournée par `calculeIntersection` était inférieure à environ 5×10^{-8}, le programme bloquait. La cause exacte de ce problème n'a pas été déterminée ; cependant, il n'est pas présent avec la nouvelle implantation de `calculeIntersection`.

Pour le calcul des segments de trajectoire à travers chaque voxel, la figure 6.5 montre que les résultats d'environ 90% des appels à la fonction contiennent une erreur entre 1×10^{-6} et 1×10^{-8} et environ 9%, une erreur supérieure à 1×10^{-6}. Bien que ces taux d'erreurs soient beaucoup plus élevés que ceux de la fonction `calculeIntersection`, ils demeurent nettement inférieurs à l'erreur minimale détectée dans les résultats finaux, qui était d'environ 1×10^{-2}. Une deuxième version de la fonction `calculeSegment` a été implantée, qui élimine entre autres certains calculs en utilisant des valeurs déjà disponibles en mémoire. Cette nouvelle implantation réduit quelque peu le taux d'erreurs entre 1×10^{-6} et 1×10^{-8} (à environ 79%), mais garde un taux d'erreurs supérieures à 1×10^{-6}, celui qui serait le plus important de réduire, identique à celui de la première version. L'avantage de cette deuxième version n'est cependant pas suffisant pour que son adoption dans bGPUMCD soit utile, étant donné son plus long temps d'exécution.

Enfin, la fonction qui calcule la rotation des vecteurs de direction des particules donne lieu à des erreurs beaucoup plus faibles que les deux autres. Le taux d'erreurs sur les coordonnées en X et en Y est plus élevé que sur l'axe Z, en raison de la méthode utilisée pour effectuer la rotation (deux changements de système de référence), qui nécessite plus de termes pour leur calcul. En ce qui a trait à l'erreur relative, comme la direction est un vecteur unitaire, elle demeure généralement très faible ; environ 2.5% des appels génèrent une erreur relative supérieure à 10^{-6} (ou 0.0001%) pour les axes X et Y, et 1.6% pour l'axe Z, et moins de 3 appels par million génèrent une erreur relative supérieure à 10^{-3} (0.1%) pour tous les axes. Une erreur relative de 10^{-6} dans les trois axes engendre une déviation maximale d'environ

6×10^{-5} degré, qui est environ 8 fois inférieure au seuil de 5×10^{-4} degré à partir duquel une erreur devient détectable dans les résultats finaux.

Le seul inconvénient de ce test est qu'il ne tient pas compte d'appels possibles à des fonctions trigonométriques étant donné que la fonction utilise directement les sinus et cosinus des angles. Cependant, dans la majorité des cas, les angles sont choisis directement à partir d'un nombre aléatoire, ce qui implique une erreur de moins d'un demi-bit lors du calcul de son sinus et cosinus, ou alors leur cosinus est sélectionné directement, ce qui exclut tout appel à une fonction trigonométrique. Ainsi, ces fonctions devraient n'avoir que peu d'effet sur la précision de la rotation.

7.2 Discrétisation

Les résultats présentés dans les tableaux 6.5 et 6.6 donnent peu d'informations sur les différences qui peuvent être causées par des erreurs numériques à différentes résolutions de voxels. Toutes les comparaisons donnent une différence très grande, qui laisserait normalement croire que les situations comparées ne sont pas les mêmes. Cette différence est beaucoup plus marquée lorsque la distribution utilisée comme référence est celle qui a la résolution la plus élevée. Cet effet est normal étant donné que la différence est calculée pour chaque voxel de la distribution référence. Lorsqu'il y a plus de voxels dans la distribution référence, les différences s'additionnent, tandis que lorsqu'il y a plus de voxels dans la distribution évaluée, les différences s'annulent.

Les seuls cas où une relativement faible différence est observée sont lorsque la résolution de la distribution comparée est un multiple de celle de la distribution référence. Le fait qu'une différence existe dans ces cas révèle la présence d'erreurs numériques. La cause la plus probable de ces erreurs est le processus d'accumulation d'énergie dans les voxels. Comme expliqué plus tôt, plus le nombre de particules passant à travers un voxel est grand, plus les contributions individuelles perdent de la précision. Lorsque le nombre de voxels est doublé le long de chaque axe, le nombre de sommations est multiplié par 8. Chacune de ces nouvelles sommations contient des erreurs d'arrondi différentes de la sommation originale. De plus, pour des résolutions plus basses, l'erreur décrite à la section 4.4.1, par laquelle les contributions trop petites par rapport au total sont éliminées, survient plus rapidement. L'ensemble de ces erreurs expliquerait les différences observées dans le tableau 6.5 pour les résolutions qui sont un multiple l'une de l'autre.

Les différences élevées dans les autres cas sont expliquées par l'accumulation d'énergie autour de la source de radiation. Une grande quantité d'énergie est déposée très près de la source, mais l'énergie diminue très rapidement avec la distance de la source. La discrétisation

en voxels peut difficilement capturer cette diminution rapide. Des discrétisations différentes donnent lieu à différents gradients d'énergie apparents.

Une autre méthode, consistant à convertir la distribution évaluée à la résolution de la distribution référence avant d'effectuer la comparaison, a été testée, avec des résultats similaires : dès que les voxels ne sont pas exactement alignés, une grande différence apparaît.

En ce qui concerne les tests avec des voxels non cubiques, deux faits ressortent particulièrement du tableau 6.6. Le premier est que, en utilisant une distribution dont la résolution est supérieure le long de l'axe Z (celui le long duquel les sources sont positionnées), la différence est dans chaque cas moins marquée qu'avec une résolution augmentée le long des autres axes. La deuxième remarque est que les comparaisons avec la distribution de $300 \times 300 \times 300$ voxels donnent une différence plus faible que les comparaisons du tableau 6.5 lorsque les résolutions sont un multiple l'une de l'autre. Cet effet est normal étant donné que le long d'un seul axe, la résolution est changée pour un multiple de la résolution initiale.

Les observations discutées dans cette section indiquent qu'il n'est pas pratique de comparer des distributions de dose de résolutions différentes avec le niveau de précision utilisé lors de ces tests. Le tableau 6.5 montre que la résolution a bel et bien un effet sur la précision des résultats, mais que cet effet dépend d'erreurs numériques déjà présentes. De plus, il n'est pas clair si l'augmentation de la résolution a un effet positif ou négatif sur ces erreurs. Par ailleurs, comme discuté à la section 7.1.2, la discrétisation en voxels semble n'avoir aucun impact remarquable sur la précision des résultats, comparée au positionnement exact des particules. Le seul impact serait donc sur le temps de calcul, qui est directement proportionnel à la racine cubique du nombre total de voxels, selon la figure 6.6.

7.3 Détection automatique

Trois aspects du module DSA seront discutés dans cette section. L'exactitude des résultats sera premièrement abordée, pour déterminer si les événements détectés correspondent en effet à des erreurs dans les calculs ; les effets de l'utilisation du module sur le temps de calcul seront ensuite expliqués ; finalement, les limitations du module seront discutées, en ce qui concerne la validité de son implantation et son utilité.

7.3.1 Exactitude des résultats

Deux méthodes sont utilisées pour vérifier le bon fonctionnement du module DSA. La première consiste à comparer ses résultats à tous ceux discutés précédemment ; s'ils concordent, il est raisonnable de conclure que le module fonctionne correctement. La seconde consiste à examiner et analyser manuellement le code source aux endroits signalés par le module pour

déterminer s'ils sont réellement à risque de causer des erreurs.

Erreurs trouvées précédemment

La première section en importance dans le tableau 6.7 concerne le calcul de la distance entre une particule et une surface quadrique et, dans une moindre mesure, une fonction qui y est souvent associée et qui sert à déterminer de quel côté d'une surface se trouve une particule. La différence la plus marquée est la diminution majeure du nombre d'événements lorsque le calcul utilise la méthode de Newton par rapport à l'algorithme initial, en concordance avec les résultats présentés aux sections 4.4.3, 6.1.2 et 6.1.3. Cependant, la nature même des événements détectés et leur interaction ont provoqué une différence beaucoup plus grande qu'elle ne l'est en réalité.

De façon générale, les opérations conditionnelles sont toutes converties pour le type `dsa_t` en une comparaison avec zéro, précédée d'une soustraction s'il y a lieu. Une erreur dans le résultat de la comparaison indique que le nombre comparé à zéro est un zéro informatique, causé par une annulation lors de la soustraction. Dans les cas détectés où la comparaison était faite directement avec zéro, le code source de `bGPUMCD` présentait également une soustraction causant une annulation dans les lignes précédant la comparaison. Une telle situation résulte en un dédoublement des erreurs détectées – une pour l'annulation et une pour la comparaison. Ce dédoublement est particulièrement visible dans `calculeIntersection`, où le calcul du radical pour résoudre une équation quadratique (ligne 4, algorithme 4.1) génère environ 2.2 millions d'annulations. L'algorithme vérifie ensuite si ce radical est positif ou négatif (ligne 5), ce qui résulte en environ 1.3 million d'erreurs d'embranchement à la suite de la comparaison. La différence entre les deux quantités est simplement une conséquence du seuil utilisé pour que le résultat d'une soustraction soit considéré comme trop imprécis. Lors de ces tests, il était fixé à 3 décimales, ce qui correspond à environ 10 bits de précision perdus. La soustraction de la ligne 4 donne lieu à une erreur d'embranchement lorsque toute la précision est perdue.

De manière similaire, l'enchaînement des opérations peut causer une augmentation artificielle du nombre d'erreurs détectées. Par exemple, un nombre peut devenir un zéro informatique à la suite d'une annulation ; les opérations subséquentes impliquant ce nombre risquent donc de causer des erreurs, possiblement à répétition. Ce problème survient dans `calculeIntersection`, où la valeur de la variable `radical` (ligne 4) est utilisée comme paramètre dans le calcul d'une racine carrée et génère donc d'autres erreurs aux lignes 8 et 9.

Même en considérant uniquement les annulations, la version analytique de `calculeIntersection` génère beaucoup plus d'événements que la méthode de Newton. La très vaste majorité de ceux-ci, comme mentionné plus haut, sont causés par le calcul du radical, qui s'approche de zéro lorsque la trajectoire d'une particule est presque tangente à la surface

avec laquelle l'intersection est calculée. Les autres proviennent en grande partie du calcul de `B` (ligne 2) et, dans une moindre mesure, des calculs de la distance entre la particule et la boîte englobante de la surface quadrique.

Les deux plus importantes sources d'erreur signalées par le module DSA dans la fonction `calculeIntersection` sont en accord avec celles décrites à la section 4.4.3. Cependant, les autres sources d'erreur décrites initialement (calcul de `C` lorsque le photon est proche de la surface, calcul des racines lorsqu'elles sont très près l'une de l'autre) n'ont pas été détectées par le module. Trois explications pourraient justifier cette divergence. Étant donné que ces erreurs étaient mentionnées d'un point de vue théorique, il est possible qu'en pratique elles n'aient pas lieu. Cependant, il est aussi possible que des annulations aient lieu, mais à un niveau moins important que le seuil de 3 décimales utilisé lors des tests. Enfin, plusieurs observations, qui seront discutées plus loin, semblent mettre en doute la fiabilité de la localisation exacte des événements par le module DSA ; des erreurs dans le calcul de `C` pourraient donc avoir été attribuées à la ligne précédente.

Nouvelles erreurs

La seule autre source spécifique d'erreur découverte par les tests décrits plus tôt (sections 4.3, 5.1 et 5.2) se trouve dans la sommation utilisée pour l'accumulation de l'énergie dans chaque voxel, qui n'est pas prise en compte par le module DSA. Tous les autres événements signalés par le module DSA doivent donc être analysés individuellement pour déterminer si des erreurs véritables en sont la cause.

Le dernier événement généré en quantité non négligeable par la version analytique de `calculeIntersection` provient du calcul de la distance avec la boîte englobante, utilisée pour limiter la longueur du cylindre, étant donné que la surface quadrique décrit un cylindre de longueur infinie. L'erreur indiquée semble se produire relativement peu fréquemment, lorsqu'une particule passe très près du rebord aux extrémités d'un cylindre.

Dans la deuxième version de `calculeIntersection`, qui utilise une méthode de descente de gradient, la presque totalité des événements détectés provient d'une annulation dans le calcul de la dérivée de la fonction à optimiser. Ce résultat est logique étant donné que l'objectif ultime de `calculeIntersection` est que cette dérivée devienne égale à zéro, ce qui nécessite qu'une série de termes calculés à partir de la position, de la direction et de la distance à parcourir de la particule ainsi que des paramètres de la surface quadrique s'annulent. Cela donne lieu presque assurément à une perte de précision de plusieurs bits. Ce calcul est presque exactement le même que celui de `B` dans l'algorithme analytique (4.1). Cependant, il résulte beaucoup plus fréquemment en une annulation (440000 par rapport à 170000) étant donné qu'il est répété jusqu'à ce qu'il s'approche de zéro – la condition nécessaire pour qu'une

annulation se produise. Les problèmes potentiels causés par cette annulation sont cependant limités par plusieurs vérifications plus loin dans l'algorithme, qui assurent que la distance trouvée place la particule directement sur son intersection avec la surface, ce qui donne la distance la plus précise possible avec des calculs en simple précision.

La seule autre différence notable entre les résultats du module DSA pour les deux versions de `calculeIntersection` se trouve dans une autre fonction, qui sert à déterminer si un point se trouve à l'intérieur d'une surface quadrique. Elle est utilisée pour trouver la source dans laquelle se trouve un certain point, afin de connaître la densité du milieu à cette position. La ligne signalée par le module DSA indique que l'erreur se produit lorsque la position du point est de zéro le long d'un axe. Ce résultat est improbable étant donné que dans la majorité des appels à cette fonction, le point donné en argument se trouve au milieu de la distance entre deux surfaces le long de la trajectoire d'une particule, qui coïncide très rarement avec une coordonnée de zéro pour un axe. Ainsi, il est possible qu'en raison d'un défaut dans le module, l'erreur se trouve véritablement à la ligne suivante, qui causerait alors une annulation lorsque le point se trouve très près d'une surface. D'autres événements semblent indiquer que la seconde ligne est la bonne : une erreur d'opération conditionnelle est également signalée à la première ligne alors qu'aucune comparaison ne s'y trouve ; la seconde ligne contient une addition de plusieurs termes en plus d'une comparaison. Des tests supplémentaires seraient nécessaires pour expliquer pourquoi la méthode de Newton cause une augmentation considérable des erreurs dans cette fonction (170000 par rapport à 50000).

Le reste des erreurs détectées par le module DSA présentent un profil très similaire pour les deux versions de `calculeIntersection`. Elles peuvent être divisée parmi trois composantes du programme : le calcul de la longueur de la trajectoire dans chaque voxel par `calculeSegment`, l'estimation du pas entre deux interactions par `calculePas` et dans la boucle principale de l'algorithme utilisé par `bGPUMCD`, qui appelle les deux fonctions précédentes et gère les interactions.

Dans `calculeSegment`, de nombreuses annulations sont présentes qui semblent se produire lorsque les particules s'approchent des frontières extérieures du volume simulé. Cette erreur ne devrait pas avoir d'impact important sur la simulation pour plusieurs raisons. De façon générale, les sources de radiation sont plutôt situées près du centre du volume pour en simuler les effets le plus complètement possible. Ainsi, les voxels près des frontières ne contiennent pas une dose significative de radiation et ne sont donc pas considérés comme importants pour la planification de traitement. Par ailleurs, il a été démontré dans la section 7.1.2 que les erreurs dans la fonction `calculeSegment` ont un impact sur le résultat d'une simulation seulement à partir d'une grande taille (0.01), qui est de beaucoup supérieure à un nombre qui résulte d'une annulation – qui est nécessairement très près de zéro.

Dans `calculePas`, la plupart des erreurs sont dans des opérations conditionnelles qui comparent les différentes distances trouvées par `calculeIntersection` avec les surfaces contenues dans le volume simulé. Le nombre plus grand d'erreurs lorsque la version analytique de `calculeIntersection` est utilisée pourrait être expliqué par la taille légèrement plus grande des erreurs dans son résultat. Une petite proportion des erreurs dans `calculePas` sont des annulations qui semblent se produire lorsque la prochaine interaction d'une particule a lieu très près d'une surface, ce qui se produit relativement peu fréquemment.

Dans la boucle principale de `bGPUMCD`, les erreurs détectées, peu fréquentes, proviennent de trois opérations différentes. La première est le calcul du voxel dans lequel se trouve une particule. Comme anticipé à la section 4.3.1, il arrive que la particule soit localisée dans le mauvais voxel. Cependant, cette erreur n'est pas suffisamment fréquente pour avoir un effet visible sur le résultat final d'une simulation, selon les résultats d'injection d'erreur déterminés plus haut (section 7.1.2) . La deuxième erreur concerne le segment dans le voxel où une particule termine sa trajectoire avant une interaction. La longueur de ce segment est calculée à partir du pas total et de la somme des segments dans tous les voxels précédents. Ainsi, si ce segment est très court (ou si l'ensemble des autres segments contient des erreurs trop grandes), une annulation se produit. D'autres tests ont révélé quand dans certains cas très rares, la longueur calculée pour ce segment est négative. Cette erreur peut être facilement diminuée en vérifiant que la distance dans le dernier voxel n'est pas négative. Cependant, comme déterminé précédemment (section 7.1.2), elle n'est pas suffisamment grande pour avoir un effet sur le résultat final. La troisième erreur se produit extrêmement rarement (seulement 5 cas détectés), dans la mise à jour de la position d'une particule lorsque sa prochaine interaction se trouve dans un des plans centrés à l'origine, c'est-à-dire qu'une des composantes de la position devient zéro. Il est improbable que cette erreur ait le moindre effet visible sur le résultat final.

7.3.2 Temps de calcul

La dernière ligne du tableau 6.7 indique l'augmentation du temps d'exécution de `bGPUMCD` lorsque le module DSA est utilisé par rapport au temps d'exécution normal. Bien que le temps d'exécution avec le module DSA n'affecte pas l'exactitude de ses résultats, il rend son utilisation routinière moins pratique en ce qui a trait au débogage de l'application testée. Le ralentissement extrême observé est causé par plusieurs facteurs : répétition des calculs, utilisation de nombres aléatoires, calculs nécessaires pour les vérifications et l'enregistrement des erreurs, mémoire supplémentaire utilisée et conversion entre les différents types.

Répétition des calculs

Chaque calcul en virgule flottante est effectué trois fois, tout comme avec la librairie CADNA de Jézéquel et Chesneaux (2008). Cependant, le ralentissement causé par l'utilisation de CADNA était limité à ce facteur. Dans un programme sur GPU, cette multiplication des calculs a des répercussions beaucoup plus importantes. La première est la plus grande quantité de mémoire nécessaire. Pour `bGPUMCD`, le nombre de photons qui peuvent être simulés simultanément est limité par la mémoire disponible sur chaque multiprocesseur de la carte graphique. Comme la plupart des calculs dans `bGPUMCD` sont faits avec des nombres à virgule flottante, l'utilisation d'un type qui occupe trois fois plus de mémoire diminue d'autant le parallélisme, en plus d'augmenter la quantité de calculs.

Nombres aléatoires

La génération d'un nombre aléatoire pour chaque opération en virgule flottante cause également un ralentissement majeur. En effet, chaque nombre aléatoire est calculé à partir de plusieurs additions et multiplications d'entiers, ainsi que d'une multiplication en virgule flottante. De plus, l'état du générateur de nombres aléatoires est préservé dans la mémoire globale plutôt que locale pour limiter les modifications à apporter au code source de `bGPUMCD`. Les accès à la mémoire globale causent également une augmentation du temps d'exécution.

Détection d'événements

Lorsque tous les types de test disponibles dans le module DSA sont activés, chaque opération impliquant un nombre à virgule flottante donne lieu à une vérification. Cette vérification nécessite de calculer la moyenne et la variance des différentes versions d'une variable représentée par le type `dsa_t`, ainsi que deux logarithmes pour connaître le nombre de bits exacts dans les variables lorsque la vérification concerne une possible annulation.

Si un test est positif, la ligne de code où l'opération a lieu est enregistrée dans un tampon. Au départ, le tampon d'erreurs est complètement vide. La première fois qu'une erreur est détectée à une certaine ligne, le numéro de cette ligne est ajouté au tampon pour y réserver un espace, qui sert à compter le nombre d'erreurs à cette ligne. Pour chaque erreur subséquente, le tampon est parcouru, jusqu'à ce que le numéro de ligne soit trouvé et le compteur correspondant incrémenté. Cette façon d'enregistrer les événements provoque deux délais. Le premier est causé par la recherche du bon numéro de ligne à travers le tampon. Le second est simplement causé par l'incrémentation du compteur, qui doit se faire de façon atomique et entraîne donc une sérialisation des opérations. Cette sérialisation se produit fréquemment, surtout pour les lignes où beaucoup d'erreurs sont enregistrées, étant donné que tous les fils

d'exécution au sein d'un *warp* sont synchronisés sur la même ligne de code.

Conversions

Lorsqu'une opération a lieu entre un nombre à virgule flottante – défini comme type `dsa_t` – et une variable d'un autre type, le second type doit parfois être converti en type `dsa_t`. Sans le module DSA, cette conversion est très rapide et peut même être faite directement par le compilateur. Avec le module, à cause de son implantation particulière ou à cause du compilateur `nvcc`, cette conversion est faite lors de l'exécution et nécessite une triple copie d'une variable, en plus de sa conversion. Dans certaines portions de `bGPUMCD`, de telles conversions ont souvent lieu et causent un ralentissement supplémentaire. C'est le cas de la fonction à optimiser avec la méthode de Newton dans `calculeIntersection`, qui doit être calculée à répétition ; c'est la cause la plus probable de la grande différence dans le temps d'exécution entre les deux algorithmes de `calculeIntersection` lorsque le module DSA est utilisé.

Taille de l'exécutable

Une dernière cause de ralentissement est la taille du programme exécutable. Toutes les opérations de base (avec des nombres à virgule flottante) sont remplacées par une longue série d'opérations, qui doivent être mises *inline* par le compilateur, ce qui résulte en un exécutable environ deux fois plus gros. Le plus grand espace occupé par le code fait que la cache contenant les instructions doit être renouvelée de façon beaucoup plus fréquente, ce qui entraîne des délais.

7.3.3 Limitations

Quatre éléments indépendants des erreurs mêmes affectent les événements détectés par le module DSA et les lignes de codes auxquelles ces événements sont attribués : les opérations atomiques ; le niveau d'optimisation ; l'association imprécise entre les lignes de code assembleur et les lignes de code source correspondantes ; des défauts dans l'implantation du module ou dans le compilateur.

Opérations atomiques

La sommation utilisée pour accumuler l'énergie dans chaque voxel s'est révélée problématique lors de l'implantation du module DSA. En effet, le mode d'arrondi affecte énormément le résultat de cette sommation en raison de la grande différence entre les valeurs ajoutées et le total. Ainsi, très rapidement au cours de la simulation, le résultat dans la plupart des

voxels devient un zéro informatique. Pour cette raison, – bien que la cause du problème soit la sommation plutôt que le fait que les opérations soient atomiques – les opérations ont été implantées de façon telle que l'addition sur chaque version de la variable dans le type `dsa_t` soit faite en arrondissant vers le nombre le plus proche, plutôt qu'aléatoirement vers le haut ou vers le bas.

Optimisation

Pour tenter d'obtenir des résultats les plus fidèles possible à une exécution normale de `bGPUMCD`, les tests effectués avec le module DSA ont été faits avec le même niveau d'optimisation – le plus élevé, qui est la valeur par défaut. L'optimisation entraîne des difficultés pour l'identification exacte des lignes de code étant donné que le compilateur peut changer l'ordre des opérations, en enlever en sauvegardant des valeurs dans des registres ou en ajouter pour diminuer le nombre de registres utilisés.

En insérant une grande quantité de calculs dans chaque opération, le module DSA empêche beaucoup de ces optimisations par le compilateur. Le programme analysé n'est donc pas exactement le même que celui qui est exécuté sans l'analyse. Il est nécessaire de tenir compte de cette divergence lors de l'interprétation des résultats obtenus. Par exemple, certaines séries d'opérations sont répétées dans la fonction `calculeIntersection` – pour plus de clarté dans le code source. Cependant, le module DSA ne signale une erreur qu'à une seule de ces lignes à la fois. Ainsi, une correction qui vise à retirer des erreurs doit envisager la possibilité que ces erreurs se produisent à d'autres endroits dans le code, qui ne seraient pas indiqués par l'analyse.

Traduction en code assembleur

Le compilateur `nvcc` traduit les lignes de code source CUDA en instructions d'assembleur pour l'architecture PTX, qui est commune à toutes les cartes graphiques NVIDIA. Pour aider au débogage, ces instructions sont entrelacées de directives spéciales qui indiquent à quelle ligne de code source correspond chaque groupe d'instructions. Comme expliqué à la section 5.4.3, il n'est pas possible d'accéder à ces informations à partir du programme compilé. C'est pourquoi le module DSA fait appel à un script qui insère les numéros de ligne dans les instructions mêmes.

Cependant, le processus d'optimisation – qui a lieu avant l'émission du code PTX – peut modifier grandement l'ordre des instructions et leur nombre. Ainsi, un certain groupe d'instructions peut correspondre à plusieurs lignes de code. Cette situation est de plus compliquée par l'insertion du code des fonctions *inline* dans le code principal, qui permet au compilateur

de mélanger les lignes de plusieurs fichiers en même temps.

Lorsque le fichier PTX a été produit par le compilateur, le script responsable de l'insertion des numéros de ligne dans les instructions parcourt ce fichier ligne par ligne. Chaque fois qu'il rencontre une directive indiquant une ligne de code, il considère les instructions suivantes comme associées à cette ligne. Il l'ignore toutefois si elle se trouve dans un fichier appartenant au module DSA, étant donné que cette information n'est pas utile au processus d'analyse. Les seuls points d'entrée dans le module DSA à partir du programme compilé sont les opérations arithmétiques et les fonctions mathématiques surchargées. Ainsi, en ignorant les instructions appartenant à ce module, le script les associe plutôt avec la ligne où l'opération a lieu dans le programme analysé. Chaque opération testée du programme analysé est accompagnée d'un appel – conditionnel – à la fonction `enregistrerErreur` dans le cas où une erreur est détectée. Les arguments de cette fonction comprennent le numéro de la ligne de code source et un numéro associé au fichier correspondant. Pour faciliter l'interaction du script avec le fichier PTX, la fonction `enregistrerErreur` n'est pas *inline*. Cela permet d'isoler clairement chaque appel ainsi que tous les paramètres de cette fonction dans les instructions en assembleur.

Le résultat final de la modification du fichier PTX est un programme dans lequel chaque appel à `enregistrerErreur` est associé avec la ligne de code la plus récemment indiquée par le compilateur, après optimisation. Lors des tests présentés ici, plusieurs lignes de code n'apparaissaient pas dans le fichier PTX, pour les raisons expliquées plus haut. En conséquence – et d'après une analyse manuelle des résultats et du code – plusieurs des événements signalés par le module DSA apparaissent de façon erronée à une ligne précédant celle où l'erreur s'est réellement produite. L'inverse n'a jamais été observé. Dans la majorité des cas, des erreurs étaient rapportées à une ligne contenant une condition, alors qu'elles se produisaient en réalité à la première ligne d'une des branches du code dépendant de cette condition. Ce résultat est probablement causé par un réarrangement des opérations suivant la condition par le compilateur.

Autres problèmes

Plusieurs autres problèmes ont été remarqués lors des tests du module DSA. Leur cause probable est une erreur dans l'implantation du module ; cependant, il est également possible que le compilateur `nvcc` en soit responsable.

Lors d'une première implantation de la méthode d'analyse par DSA, les fonctions surchargées recevaient leurs paramètres par référence. Dans les cas où les opérandes sont tous de type `dsa_t`, cette façon de faire fonctionnait parfaitement. Cependant, dans certaines opérations conditionnelles, où une variable `dsa_t` est comparée avec un nombre entier (0 ou 1), le

compilateur n'utilisait pas la fonction surchargée. C'était plutôt la variable de type `dsa_t` qui était convertie en nombre entier avant d'effectuer la comparaison. Ainsi, en plus d'une erreur dans l'insertion des vérifications, une erreur était introduite dans les résultats du programme analysé. Ce problème a complètement disparu après que le module ait été modifié pour que les paramètres de toutes les fonctions soient passés par valeur plutôt que par référence.

Un autre problème, possiblement relié au compilateur, concerne la mise en ligne des fonctions. Par défaut, toutes les fonctions CUDA sont considérées *inline* par le compilateur. Sans le module DSA, `bGPUMCD` se compile et s'exécute de cette façon sans problème. Cependant, lorsque le module d'analyse est inclus sans modifier le programme principal, le kernel ne s'exécute plus, avec une erreur indiquant la possibilité d'un problème de mémoire. L'erreur disparaît lorsque la fonction `calculeIntersection` est compilée avec l'attribut `noinline`, qui la garde séparée du code principal.

Finalement, lors des tests d'analyse, le programme bloquait parfois, sans message d'erreur et sans terminer son exécution. Il est possible que la fonction `enregistrerErreur` soit à l'origine de ce problème étant donné qu'il ne se produit pas lorsque tous les tests du module DSA sont désactivés. Cependant, des tests supplémentaires sont nécessaires pour en déterminer la cause exacte.

Tous ces problèmes rendent moins fiables les résultats du module DSA. Une analyse approfondie de ces résultats est nécessaire pour les valider et il n'est pas garanti que toutes les erreurs numériques aient été détectées. Cependant, les résultats du tableau 6.7, interprétés et discutés ci-haut, corroborent particulièrement bien ceux des méthodes présentées plus tôt, surtout en ce qui concerne les fonctions `calculeIntersection` et `calculeSegment`. À défaut de prouver le fonctionnement parfait du module DSA, cela démontre l'utilité de cette méthode pour détecter les erreurs numériques.

CHAPITRE 8

CONCLUSION

8.1 Impact des erreurs numériques

Cette étude visait à déterminer si des erreurs reliées à la représentation en virgule flottante étaient présentes dans `bGPUMCD`, d'en évaluer l'impact et d'implanter une méthode automatique pour les détecter. Ces objectifs ont été atteints par les travaux présentés et ont permis de vérifier les hypothèses émises plus tôt.

L'hypothèse 1, qui affirmait la possibilité de quantifier l'apport individuel de trois causes générales d'erreur, est partiellement vraie. En effet, bien que la précision des calculs et l'implantation logicielle ou matérielle des fonctions mathématiques fournies aient été évaluées séparément, l'impact de la discrétisation en voxel a été plus difficile à déterminer spécifiquement. Les tests d'injection d'erreur ont cependant établi que cet impact est relativement faible par rapport à d'autres sources d'erreur.

Les hypothèses 2 et 3, qui proposaient une limite supérieure aux erreurs observables, sont toutes deux fausses dans leur formulation présente. L'erreur causée par l'accumulation d'énergie pouvait en effet atteindre plus de 75% dans certains voxels, plutôt que le 1% envisagé. En ajoutant qu'elle concerne uniquement les erreurs reliées au système de traçage de particules, l'hypothèse 3 peut être considérée comme vraie, avec une borne beaucoup plus serrée que 10%, étant donné qu'aucune erreur supérieure à 0.8% n'a été détectée ailleurs que dans le processus d'accumulation d'énergie.

L'hypothèse 4 a été confirmée par l'implantation du module DSA, qui a permis de détecter les erreurs découvertes par d'autres moyens. Bien qu'il soit encore incomplet, le module DSA offre la preuve que la détection automatique d'erreurs numériques sur GPU est possible.

8.2 Travaux futurs

Des solutions ont déjà été apportées aux erreurs trouvées dans `bGPUMCD` et il a été établi que ce programme ne contient pas d'autres erreurs numériques majeures. Ainsi, cette partie du travail peut être considérée comme complétée.

Le module DSA, quant à lui, est encore inachevé. Plusieurs opérations restent à implanter pour qu'il puisse détecter un plus grand nombre d'erreurs. D'autres changements sont requis pour que ce module soit utilisable plus facilement par d'autres programmes que `bGPUMCD`, sans

requérir de modifications au programme analysé – autre que les changements de type. Finalement, des corrections sont nécessaires pour assurer que le module n'empêche pas l'exécution du programme testé et pour réduire son impact sur le temps de calcul.

RÉFÉRENCES

AGOSTINELLI, S. *ET AL.* (2003). Geant4—a simulation toolkit. *Nuclear Instruments and Methods in Physics Research*, 506, 250 – 303.

BARÓ, J., SEMPAU, J., FERNÁNDEZ-VAREA, J. et SALVAT, F. (1995). PENELOPE : An algorithm for Monte Carlo simulation of the penetration and energy loss of electrons and positrons in matter. *Nuclear Instruments and Methods in Physics Research Section B : Beam Interactions with Materials and Atoms*, 100, 31 – 46.

BENZ, F., HILDEBRANDT, A. et HACK, S. (2012). A dynamic program analysis to find floating-point accuracy problems. *ACM SIGPLAN Notices*. vol. 47, 453 – 462.

BLANCHET, B., MAUBORGNE, L., COUSOT, P., MINE, A., COUSOT, R., MONNIAUX, D., FERET, J. et RIVAL, X. (2003). A static analyzer for large safety-critical software. *ACM SIGPLAN Notices*. San Diego, CA, United states, vol. 38, 196 – 207.

BRISEBARRE, N., JOLDES, M., KORNERUP, P., MARTIN-DOREL, E. et MULLER, J.-M. (2011). Augmented precision square roots and 2-D norms, and discussion on correctly rounding sqrt(x^2+y^2). *Proceedings - Symposium on Computer Arithmetic*. Tubingen, Germany, 23 – 30.

CARTER, L. L., CASHWELL, E. D. et TAYLOR, W. M. (1972). Monte Carlo sampling with continuously varying cross sections along flight paths. *Nuclear Science and Engineering*, 48, 403 – 411.

CASTALDO, A. M., WHALEY, R. C. et CHRONOPOULOS, A. T. (2009). Reducing floating point error in dot product using the superblock family of algorithms. *SIAM journal on scientific computing*, 31, 1156 –1174.

COUSOT, P. et COUSOT, R. (1977). Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA, POPL '77, 238–252.

DE DINECHIN, F., LAUTER, C. et MELQUIOND, G. (2011). Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Transactions on Computers*, 60, 242 – 253.

DE FIGUEIREDO, L. et STOLFI, J. (2004). Affine arithmetic : Concepts and applications. *Numerical Algorithms*, 37, 147–158.

D'SILVA, V., HALLER, L., KROENING, D. et TAUTSCHNIG, M. (2012). Numeric bounds analysis with conflict-driven learning. C. Flanagan et B. König, éditeurs, *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, vol. 7214 de *Lecture Notes in Computer Science*. 48–63.

FADDEGON, B. A., KAWRAKOW, I., KUBYSHIN, Y., PERL, J., SEMPAU, J. et URBAN, L. (2009). The accuracy of EGSnrc, Geant4 and PENELOPE Monte Carlo systems for the simulation of electron scatter in external beam radiotherapy. *Physics in Medicine and Biology*, 54, 6151–6163.

FILLIÂTRE, J.-C. et MARCHÉ, C. (2007). The Why/Krakatoa/Caduceus platform for deductive program verification. W. Damm et H. Hermanns, éditeurs, *Computer Aided Verification*, Springer Berlin Heidelberg, vol. 4590 de *Lecture Notes in Computer Science*. 173–177.

FOUSSE, L., HANROT, G., LEFÈVRE, V., PÉLISSIER, P. et ZIMMERMANN, P. (2007). MPFR : A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33, 1–14.

GOLDBERG, D. (1991). What every computer scientist should know about floating-point arithmetic. *Computing Surveys*, 23, 5–48.

GOUBAULT, E. (2001). Static analyses of the precision of floating-point operations. *Static Analysis. 8th International Symposium, SAS 2001. Proceedings*. Berlin, Germany, 234 – 59.

GOUBAULT, E. et PUTOT, S. (2006). Static analysis of numerical algorithms. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Seoul, Korea, Republic of, vol. 4134 LNCS, 18 – 34.

GOUBAULT, E., PUTOT, S., BAUFRETON, P. et GASSINO, J. (2008). Static analysis of the accuracy in control systems : Principles and experiments. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Berlin, Germany, vol. 4916 LNCS, 3 – 20.

GRANVILLIERS, L. et BENHAMOU, F. (2006). RealPaver : An interval solver using constraint satisfaction techniques. *ACM TRANS. ON MATHEMATICAL SOFTWARE*, 32, 138–156.

HARRISON, J. (2000). Formal verification of floating point trigonometric functions. *Formal Methods in Computer-Aided Design : Third International Conference FMCAD 2000, volume 1954 of Lecture Notes in Computer Science*. 217–233.

HIGHAM, N. J. (2002). *Accuracy and Stability of Numerical Algorithms*. SIAM.

HISSOINY, S., D'AMOURS, M., OZELL, B., DESPRÉS, P. et BEAULIEU, L. (2012). Sub-second high dose rate brachytherapy Monte Carlo dose calculations with bGPUMCD. *Medical Physics*, 39, 4559–4567.

HISSOINY, S., OZELL, B., BOUCHARD, H. et DESPRÉS, P. (2011a). GPUMCD : A new GPU-oriented Monte Carlo dose calculation platform. *Medical Physics*, 38, 754–764.

HISSOINY, S., OZELL, B., DESPRÉS, P. et CARRIER, J.-F. (2011b). Validation of GPUMCD for low-energy brachytherapy seed dosimetry. *Medical Physics*, 38, 4101–4107.

IEEE (2008). IEEE-754, Standard for Floating-Point Arithmetic.

IVANCIC, F., GANAI, M. K., SANKARANARAYANAN, S. et GUPTA, A. (2010). Numerical stability analysis of floating-point computations using software model checking. *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2010*. Grenoble, France, 49 – 58.

JIA, X., GU, X., GRAVES, Y. J., FOLKERTS, M. et JIANG, S. B. (2011). GPU-based fast Monte Carlo simulation for radiotherapy dose calculation. *Physics in Medicine and Biology*, 56, 7017–7031.

JIANG, D. et STEWART, N. F. (2006). Backward error analysis in computational geometry. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Glasgow, United kingdom, vol. 3980 LNCS, 50 – 59.

JOHANSSON, F. *ET AL.* (2013). *mpmath : a Python library for arbitrary-precision floating-point arithmetic (version 0.18)*. `http://mpmath.org/`.

JÉZÉQUEL, F. et CHESNEAUX, J.-M. (2008). CADNA : a library for estimating round-off error propagation. *Computer Physics Communications*, 178, 933–955.

KARAISKOS, P., PAPAGIANNIS, P., SAKELLIOU, L., ANAGNOSTOPOULOS, G. et BALTAS, D. (2001). Monte Carlo dosimetry of the selectSeed ^{125}I interstitial brachytherapy seed. *Medical Physics*, 28.

KAWRAKOW, I. (2000). Accurate condensed history Monte Carlo simulation of electron transport. ii. application to ion chamber response simulations. *Medical Physics*, 27, 499–513.

KAWRAKOW, I. et FIPPEL, M. (2000). Investigation of variance reduction techniques for Monte Carlo photon dose calculation using XVMC. *Physics in Medicine and Biology*, 45, 2163–2184.

LAM, M. O., HOLLINGSWORTH, J. K. et STEWART, G. (2012). Dynamic floating-point cancellation detection. *Parallel Computing*.

LI, W., SIMON, S. et KIESS, S. (2011). On the numerical sensitivity of computer simulations on hybrid and parallel computing systems. *Proceedings of the 2011 International Conference on High Performance Computing and Simulation, HPCS 2011*. Istanbul, Turkey, 510 – 516.

LI, W., SIMON, S. et KIESS, S. (2012). On the estimation of numerical error bounds in linear algebra based on discrete stochastic arithmetic. *Applied Numerical Mathematics*, 62, 536 – 555.

LINDERMAN, M. D., HO, M., DILL, D. L., MENG, T. H. et NOLAN, G. P. (2010). Towards program optimization through automated analysis of numerical precision. *Proceedings of the 2010 CGO - The 8th International Symposium on Code Generation and Optimization*. Toronto, ON, Canada, 230 – 237.

MARTEL, M. (2007). Semantics-based transformation of arithmetic expressions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Kongens Lyngby, Denmark, vol. 4634 LNCS, 298 – 314.

MARTEL, M. (2012). RangeLab : A static-analyzer to bound the accuracy of finite-precision computations. *Proceedings - 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2011*. Timisoara, Romania, 118 – 122.

MICHEL, C. (2002). Exact projection functions for floating point number constraints. *Seventh international symposium on Artificial Intelligence and Mathematics*.

NAKAYAMA, T. et TAKAHASHI, D. (2011). Implementation of multiple-precision floating-point arithmetic library for GPU computing. *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*. Dallas, TX, United states, 343 – 349.

NGUYEN, T. M. T. et MARCHÉ, C. (2011). Hardware-dependent proofs of numerical programs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. vol. 7086 LNCS, 314 – 329.

NVIDIA (2012). *CUDA C Programming Guide*. NVIDIA, cinquième édition.

PONSINI, O., MICHEL, C. et RUEHER, M. (2012). Refining abstract interpretation based value analysis with constraint programming techniques. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Quebec City, QC, Canada, vol. 7514 LNCS, 593 – 607.

ROKNE, J. G. (2001). Interval arithmetic and interval analysis : an introduction. *Granular computing*. Physica-Verlag GmbH, 1–22.

SIDDON, R. L. (1985). Fast calculation of the exact radiological path for a three-dimensional CT array. *Medical Physics*, 12, 252–255.

TANG, E., BARR, E., LI, X. et SU, Z. (2010). Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. *Proceedings of the 19th international symposium on Software testing and analysis*. 131–142.

VIGNES, J. (2004). Discrete stochastic arithmetic for validating results of numerical software. *Numerical Algorithms*, 37, 377–390.

WHITEHEAD, N. et FIT-FLOREA, A. (2011). Precision & performance : Floating point and IEEE 754 compliance for NVIDIA GPUs.

WILLIAMSON, J. F. (1987). Monte Carlo evaluation of kerma at a point for photon transport problems. *Medical Physics*, 14, 567–576.